Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Storm IDS: um Sistema de Detecção de Intrusão Escalável e Distribuído

João Francisco Gonçalves Sobrinho Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. João José Costa Gondim

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. Ricardo Zelenovsky

Banca examinadora composta por:

Prof. João José Costa Gondim (Orientador) — CIC/UnB
Prof. Dr. Robson de Oliveira Albuquerque — ENE/UnB
Prof. Dr. André Costa Drummond — CIC/UnB

CIP — Catalogação Internacional na Publicação

Gonçalves Sobrinho Júnior, João Francisco.

Storm IDS: um Sistema de Detecção de Intrusão Escalável e Distribuído / João Francisco Gonçalves Sobrinho Júnior. Brasília : UnB, 2016.
71 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. Sistema Detector de Intrusões, 2. Snort, 3. Storm, 4. Kafka,
5. Cassandra

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Storm IDS: um Sistema de Detecção de Intrusão Escalável e Distribuído

João Francisco Gonçalves Sobrinho Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. Robson de Oliveira Albuquerque
ENE/UnB

Prof. Dr. André Costa Drummond
CIC/UnB

Prof. Dr. Ricardo Zelenovsky
Coordenador do Curso de Engenharia da Computação

Brasília, 29 de Janeiro de 2016

Dedicatória

Dedico este trabalho a todos que me ajudaram nesta jornada!

Most coders think debugging software is about fixing a mistake, but that's wrong. Debugging's actually all about finding the bug, about understanding why the bug was there to begin with, about knowing that its existence was no accident. It came to you to deliver a message, like an unconscious bubble floating to the surface, popping with a revelation you've secretly known all along.

–Elliot Alderson

Agradecimentos

Agradeço aos meus pais que me ensinaram, guiaram, estimularam e me motivaram desde o momento que cheguei ao mundo. Sem o apoio e de vocês eu jamais poderia ter me tornado a pessoa que me tornei. Obrigado por estarem sempre ao meu lado e me fornecerem todas as ferramentas necessárias para que eu possa trilhar o meu próprio caminho.

Agradeço aos meus amigos da UnB e do Diocesano que, junto comigo, passaram por todos os apertos, decepções, desafios e alegrias no decorrer do curso.

Agradeço aos meus amigos da Scytl que me apoiaram durante todo o processo e que me cederam seus espaços de trabalho para eu poder realizar diversos testes.

Agradeço aos meus professores e mestres ao longo da vida que me ensinaram e me estimularam a seguir este caminho. Agradeço, em especial, ao meu orientador, João Gondim, por todas as suas orientações e por ter me possibilitado realizar este trabalho.

Agradeço à Universidade de Brasília por me proporcionar essa experiência única que é a graduação.

Resumo

O sistema detector de intrusões é um *software* capaz de identificar, em tempo real, o uso não autorizado ou abuso de um sistema por intrusos tanto locais como externos. Entretanto, no cenário atual, se torna cada vez mais difícil encontrar ameaça em meios que trafegam volumes imensos de dados. Esta monografia apresenta uma arquitetura escalável capaz de coletar dados de rede de várias maquinas diferentes e processá-los em ambiente distribuído. A fim de atingir esse objetivo, foram adaptados dois programas que fazem parte de um *framework* para monitoramento de sistemas distribuídos em tempo real baseado no *Apache Storm* [28] [43]: o *Resource Monitor*, que é responsável por capturar dados da máquina no qual ele está sendo executado, e o *Storm Engine*, que é encarregado por processar os dados em um ou mais computadores. Neste projeto foram criadas funcionalidades de detecção de intrusão, começando pelo *Resource Monitor* adicionando a captura de pacotes e no *Storm Engine*, criou-se um detector de intrusão. Para se encontrar vulnerabilidades, são necessárias assinaturas de ataques que possuem padrões que podem ser combinados com os pacotes capturados. Com isso, foram utilizadas as assinaturas do Snort [37], um sistema detector de intrusão livre tido como referência em detecção de intrusão. Esta arquitetura será apresentada em detalhes, evidenciando suas vantagens e desvantagens. Foram realizados testes em cada módulo do sistema e também na solução completa para demonstrar seu funcionamento e suas limitações.

Palavras-chave: Sistema Detector de Intrusões, Snort, Storm, Kafka, Cassandra

Abstract

A intrusion detection system is a software or hardware able to identify in real time the unauthorized use or abuse of a system by local and external intruders. However, nowadays it is more difficult to find threats in bulk network data. This article presents a architecture able to collect network data from many different machines and process them in a high performance cluster. Therefore, this project adapted two pieces of software that are components of a real time distributed system monitor based on Apache Storm [28] [43]: The *Resource Monitor*, witch is responsible for capture data from it host machine, and the *Storm Engine*, witch is responsible for process the data in one or more computers. In the project were created the intrusion detection functionalities, stating by the Resource Monitor adding packet capture and later in the Storm Engine it was created a intrusion detecting engine. To find vulnerabilities, signatures of pattern of attacks are needed so it can be can matched with the captured packets. Therefore were used the Snort's signatures [37], a open source intrusion detection system which is reference in intrusion detection. This architecture will be presented in details, pointing its advantages and disadvantages. Tests were performed in each module of this system and then in the hole solution to demonstrate its functions and limitations.

Keywords: Intrusion Detection System, Snort, Storm, Kafka, Cassandra

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O principal objetivo da detecção de intrusão é identificar em tempo real, o uso não autorizado ou abuso de um sistema por intrusos tanto locais como externos. Porém, este problema se torna cada vez mais complexo pela grande quantidade de dados gerados por sistemas distribuídos em vários computadores com arquiteturas diferentes.

Como uma proposta para solucionar este problema, criou-se a ideia do *Sistema Detector de Intrusão*. Estes sistemas vasculham por diversas fontes de informação a fim de detectar atividades maliciosas. Entretanto, esse processo se torna cada vez mais difícil com o aumento do volume de dados que devem ser processados. Em 2013, foi anunciado pelo então Chief Executive Officer (CEO) da *Microsoft*, Steve Ballmer, que a empresa possui mais de um milhão de servidores, disponibilizando uma grande variedade de serviços que são acessados por milhões de pessoas diferentes [30]. A quantidade de dados gerados por um sistema deste porte é imensa e as soluções tradicionais de detecção de intrusão são incapazes de trabalhar em um ambiente dessa proporção.

O processamento realizado por um detector de intrusão é muito intenso. Quando tem-se milhares de pacotes trafegando em uma rede é necessário inspecionar cada um deles para poder encontrar ameaças com precisão. Este processo necessita uma solução que tenha alto poder de computação e consiga escalar com o aumento da carga.

As principais soluções de detecção de intrusão atuais tidas como padrões de fato, como o Snort [37] e o Suricata [13], não foram projetados para atuar em ambientes com alto índice de distribuição podendo se tornar gargalos em uma rede. Neste projeto, será proposta uma arquitetura capaz de atuar em ambientes distribuídos com escalabilidade em relação ao volume de informações.

1.1 Problema

O problema de interesse neste trabalho diz respeito a: detectar intrusões em ambientes computacionais distribuídos em larga escala com escalabilidade, considerando como fonte primária fluxos de pacote de rede em grande volume, vazão e variedade.

1.2 Objetivo

1.2.1 Principal

Propor e implementar um sistema escalável capaz de detectar em larga escala atividades maliciosas em ambiente distribuído.

1.2.2 Específicos

- Propor uma solução escalável.
- Utilizar tecnologias reconhecidas.

1.3 Contribuições

Com o objetivo de detectar intrusões em ambientes distribuídos, foi planejada uma arquitetura capaz de capturar dados de diversas máquinas em uma rede e enviá-los para um *cluster* que tem grande poder de processamento.

Portanto, foi adaptada a solução de [43], que é um *framework* para monitoramento de sistemas distribuídos em tempo real baseado no *Apache Storm* [28], uma ferramenta para criação de aplicações distribuídas. Este projeto consiste de dois programas: o *Resource Monitor*, que é responsável por capturar dados da máquina no qual ele está executando, e o *Storm Engine* que é o encarregado de processar os dados em uma ou mais máquinas.

Criou-se, então, um sistema detector de intrusões baseado nas soluções citadas acima: o *Storm IDS*. Este novo projeto é composto de cinco módulos, Figura 1.1, que em conjunto são capazes de alertar atividades maliciosas.

O primeiro módulo é o de *Captura e Preprocessamento de Pacotes* que consiste de um *plugin* no *Resource Monitor*. Esse é capaz de capturar pacotes que entram e saem do computador monitorado. Após a captura, esses dados são enviados para o *Storm Engine* para realizar a detecção.

O segundo componente é o *Parser das Regras do Snort*. Para a realização da detecção de intrusão, o sistema precisa de assinaturas de ataques possibilitando fazer comparações com os pacotes. O Snort [37] é um *Intrusion Detection System (IDS)* comumente utilizado

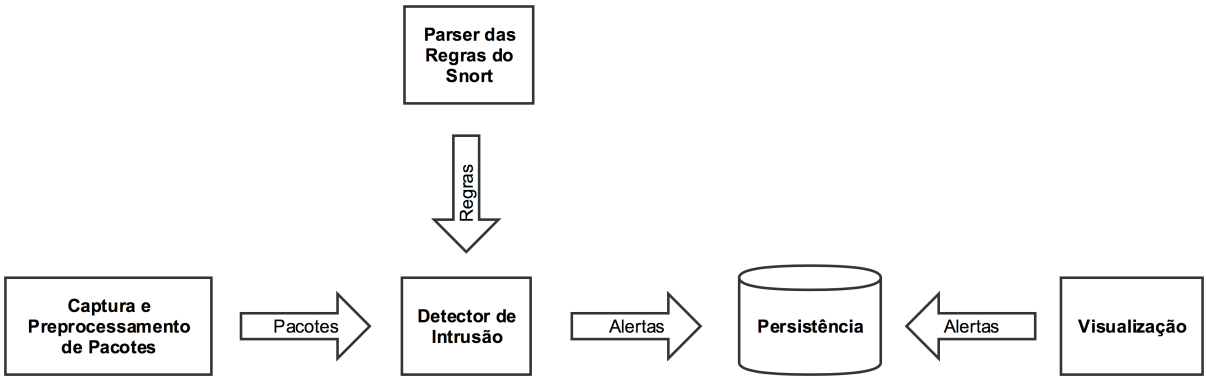

Figura 1.1: Módulos do Storm IDS.

e é tido como parâmetro para vários outros IDS. Por esse motivo, utilizou-se as assinaturas do Snort como referência.

O terceiro módulo é o *Detector de Intrusão*. Este é o coração do *Storm IDS*, pois nele são identificadas as vulnerabilidades. Ao receber os pacotes das máquinas monitoradas pelo *Resource Monitor*, o detector de intrusão utiliza o Storm para distribuir este processamento entre os computadores e *threads* disponíveis.

O quarto componente é o de *Persistência*, que é responsável por salvar os alertas gerados pelo detector de intrusão em um banco de dados.

O último módulo é o de *Visualização*. Esse é o componente que permite ao usuário visualizar os registros gerados pelo módulo de persistência.

1.4 Organização do Texto

Este capítulo introduziu de maneira geral o problema de interesse deste projeto. No próximo capítulo serão descritas em detalhes as ferramentas e tecnologias que esse projeto utiliza. No Capítulo 3 será apresentada a arquitetura do sistema e descrito com detalhes cada um dos módulos que compõem a solução. No Capítulo 4 serão apresentados os testes realizados nos sistemas e seus resultados serão analisados com detalhes. Por último, no Capítulo 5, terá uma breve conclusão dos principais tópicos discutidos no documento e dos resultados obtidos no projeto, além de apresentar trabalhos futuros para aprimorar este trabalho.

Capítulo 2

Conceitos Básicos

Este capítulo descreve os conceitos básicos utilizados neste trabalho.

2.1 Intrusion Detection Systems

Todo sistema, programa ou pessoa que tenta ou consegue acessar informações de alheios ou realizar atividades ilegais em *software* de terceiros pode ser considerado um intruso [17]. Intrusão é um conjunto de ações que buscam comprometer a integridade, confidencialidade ou disponibilidade de um recurso computacional [47]. Já o ato de detectar ações que podem comprometer a integridade, confidencialidade ou disponibilidade de um recurso computacional pode ser denominado detecção de intrusão [47].

Sistemas de detecção de intrusão são projetados para reconhecer tentativas de intrusão, bloquear ataques e produzir alertas que podem ser analisados posteriormente [6].

2.1.1 Classificação de um Detector de Intrusão

Um IDS pode ser classificado em duas categorias principais:

Host Intrusition Detection System (HIDS): Os HIDSs processam informações achadas em um ou mais sistemas, incluindo dados do sistema operacional e arquivos de programas [34]. Eles são geralmente localizados em servidores ou *workstations* e os dados são processados no nível de aplicação [2]. Podemos dizer também que os HIDSs são mais eficientes para deter ataques de origem interna, enquanto são menos eficientes para deter intrusos externos [23].

Network Intrusition Detection System (NIDS): Os NIDSs processam informações capturadas, analisando o fluxo de pacotes que trafegam pela rede [34]. Eles são geralmente posicionados em locais estratégicos da infraestrutura da rede. O NIDS

pode capturar e analisar dados para detectar ataques conhecidos pela comparação de padrões e assinaturas em um banco de dados ou pela detecção de atividades ilegais. Como esta categoria de IDS captura todos os pacotes que trafegam na rede, o NIDS é também referenciado como *packet-sniffers* [2].

2.1.2 Componentes de um Sistema Detector de Intrusão

Um IDS, em geral, consiste de três componentes funcionais [8], como ilustrado na Figura 2.1.

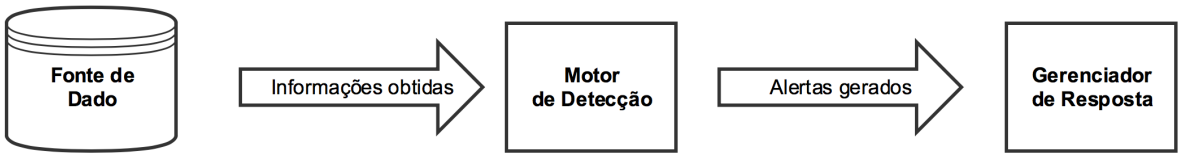

Figura 2.1: Principais componentes de um IDS adaptado de [3].

O primeiro componente, também conhecido como gerador de eventos, é o *data source* ou fonte de dados [8]. As fontes de dados são as referências que o IDS possui para achar vulnerabilidades. Elas podem ser classificadas em duas categorias.

Host-based monitors: Coleta as informações de fontes internas de um computador, geralmente a nível de sistema operacional. As fontes incluem pista de auditoria do sistema operacional e logs do sistema [3].

Network-based monitors: A fonte de dados são os pacotes da rede. Isto é usualmente alcançado usando dispositivos que são configurados para capturar todo o tráfego acessível da rede [3].

O segundo componente do sistema de detecção de intrusão é o motor de análise ou motor de detecção [8]. Este componente usa as informações da fonte de dados para detectar atividades maliciosas e pode funcionar em uma das seguintes formas de abordagens:

Detecção baseada em Assinaturas: *Signature-Based Detection* ou Detecção baseada em assinaturas é o tipo de detecção que encontra intrusões que seguem um padrão de ataque que exploram vulnerabilidades de software conhecidas (assinaturas) [25]. A maior limitação desta abordagem é que ela apenas procura por vulnerabilidades conhecidas e não consegue detectar ataques desconhecidos. [24].

Detecção baseada em Anomalias: *Anomaly based detection* ou Detecção baseada em anomalias é o tipo de detecção que procura por atividades suspeitas analisando o

comportamento da rede [24]. Nesta abordagem, eventos do sistema são analisados usando técnicas estatísticas para procurar padrões de atividades que parecem ser anormais. A principal desvantagem deste método é seu alto consumo de recursos computacionais e sua baixa eficiência em reconhecer intrusões, pois a quantidade de dados é insuficiente, gerando vários falsos positivos.

O terceiro componente de um sistema de detecção de intrusão é o *response manager* ou gerenciador de reposta [8]. Ele é responsável por armazenar e informar ao usuário quando um possível ataque for identificado.

O componente mais importante do IDS é o motor de detecção, pois ele é o responsável de fazer a detecção de atividades maliciosas. A detecção baseada em assinaturas é geralmente mais precisa que do que a baseada em anomalias, porém é necessário um processamento consideravelmente mais intenso para realizá-la.

2.1.3 Posição do IDS na topologia da rede

A posição do IDS na topologia da rede tem um grande impacto na eficiência da infraestrutura. Dependendo da estrutura da rede, o IDS pode ser colocado em um ou mais pontos [36]. Sua posição também é afetada pelo tipo de ameaças que devem ser prevenidas, podendo ser interna, externa ou ambas [36]. Por exemplo, se apenas devem ser detectadas atividades maliciosas vindo de fora da rede e há a apenas um roteador para toda a estrutura, o lugar mais indicado para posicionar o IDS é logo após o roteador ou *firewall*. Entretanto, se existem múltiplos caminhos que dão acesso a internet, o sistema detector de intrusões deve ser colocado em todos os pontos de entrada. Já no caso em que é necessária, também, a detecção de ameaças internas, o IDS deveria ser colocado em todos os pontos da rede, o que é inviável, visto que um IDS exige muito processamento. Em geral, IDS são posicionados em locais estratégicos onde os dados daquela parte da rede são muito sensíveis [36]. A Figura 2.2 mostra a posição típica de um IDS.

Tentar proteger uma rede como na Figura 2.2 é intuitivo, pois, teoricamente, os dados que chegarem no roteador são seguros. Entretanto, o IDS suporta um volume muito menor de tráfego do que um roteador, logo, eles podem se tornar excelentes alvos para ataque DoS, desabilitando todo o resto da rede.

2.2 Snort

Snort[37] é um NIDS livre e de código aberto e considerado um dos melhores disponíveis atualmente segundo [36]. Este IDS é primariamente baseado em regras, capturando dados

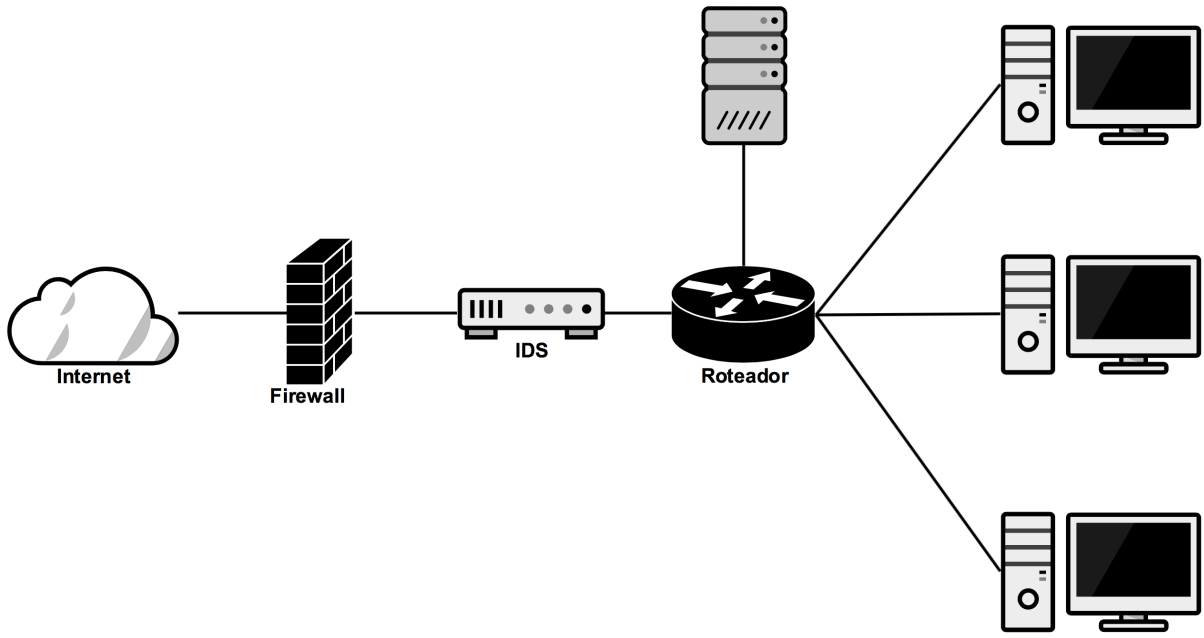

Figura 2.2: Típica posição para um Sistema Detector de Intrusões.

da rede e aplicando regras nos pacotes. Porém, com alguns plugins, pode ser realizada a detecção de anomalias na rede.

As regras do Snort são armazenadas em arquivos de texto onde cada uma representa uma categoria. Na versão 2.9.7.6 das regras do Snort, temos pouco mais de 7000 regras ativas que devem ser processadas em cada pacote. Este número pode aumentar ou diminuir de acordo com as necessidades do ambiente monitorado, pois o Snort permite que regras sejam criadas pelos seus usuários. Quando o Snort é iniciado, estas regras são transformadas em estruturas de dados para serem aplicadas aos dados capturados. Procurar regras e usá-las é um trabalho complexo, pois quanto mais regras são usadas, mais processamento é necessário para possibilitar resultados em tempo real.

2.2.1 Componentes do Snort

O Snort é dividido nos seguintes componentes principais [36]: o *packet decoder*, *preprocessors*, *detection engine* e o *logging and alerting system*. A Figura 2.3 mostra como esses componentes são dispostos [36]. Todo tráfego vindo da internet é decodificado no *Packet Decoder*. Em seguida, é realizado um pré processado no *Preprocessors* e direcionado para *Detection Engine*. Dependendo do resultado, esse tráfego será descartado, registrado ou será gerado um alerta. No *Output Modules*, esse alerta ou log será evidenciado ao usuário.

Packet Decoder

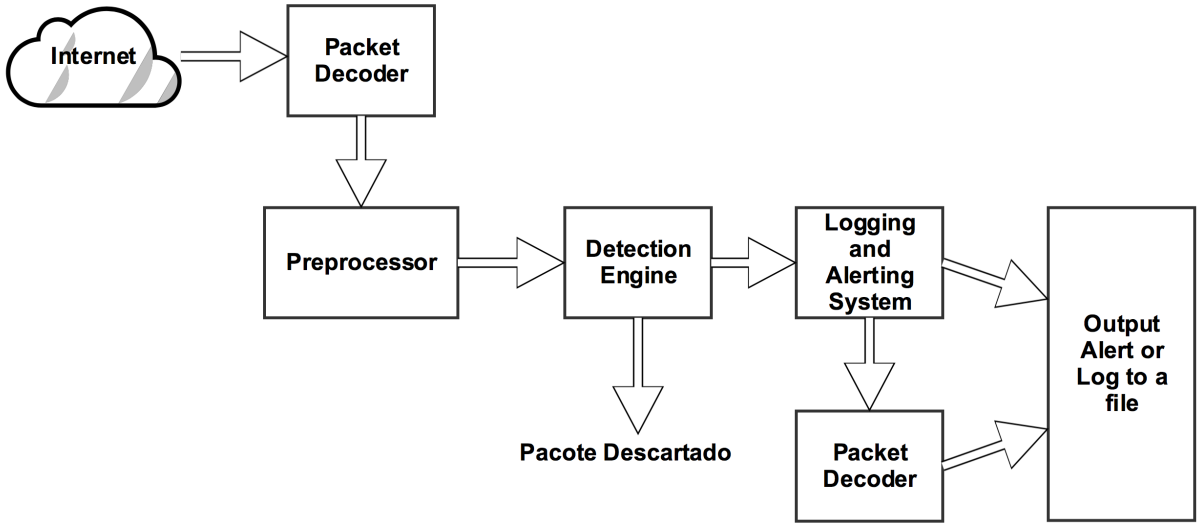

Figura 2.3: Componentes do Snort.

O *Packet Decoder* captura pacotes de diferentes interfaces e os prepara para serem preprocessados ou enviados diretamente para a unidade responsável pela detecção de ameaças. Esse comportamento pode ser configurado no arquivo de configuração [36]. Desempenho é o foco neste componente, sendo assim, a maioria das funcionalidades do *Packet Decoder* consistem em configurar ponteiros nos dados do pacote para serem analisados posteriormente no motor de detecção [38].

Preprocessors

Preprocessors são componentes ou *plugins* que podem ser usados com o Snort para modificar ou rearranjar os pacotes antes da detecção de ameaças [36]. Alguns preprocessadores podem achar anomalias nos cabeçalhos do pacote e gerar alertas. Em geral, são intrusos usando diferentes abordagens para tentar enganar o IDS. Por exemplo, para encontrar a assinatura "scripts/iisadmin"em um pacote HTTP, se esta string está sendo combinada com o padrão exato, o intruso pode fazer pequenas modificações para manter-se despercebido:

"scripts/./iisadmin"

"scripts/examples/../iisadmin"

Hackers podem piorar a situação incluindo na URI caracteres hexadecimais ou unicodes que são perfeitamente legais enquanto o servidor web está conectado. Considerando que o servidor pode entender todas essas *strings* e elas serão lidas como "string/iisadmin", o atacante será bem sucedido. Para evitar que isso aconteça, o preprocessador pode rearranjar a *string* para que essa assinatura seja detectada.

Preprocessors também podem ser utilizados para remontar fragmentos de pacotes. Quando um fluxo grande de dados entra numa rede, os pacotes são geralmente quebrados. Por exemplo, no protocolo *Ethernet v2* o tamanho máximo de um pacote é de 1500 *bytes*. Este valor é definido pelo Maximum Tranfer Unit (MTU) da rede. No Snort, é possível reorganizar estes fragmentos e recuperar os pacotes originais. Este processo é importante porque parte de uma regra pode se encontrar em um pacote e o resto em outro pacote e um *hacker* experiente pode usar da fragmentação para enganar o IDS.

Os preprocessadores são usados para se proteger contra esses ataques. Neles é possível remontar os pacotes, decodificar URIs, HTTP, reconstruir segmentos TCP, etc. Estas funcionalidades são importantes para a detecção de intrusão.

Detection Engine

O mecanismo de detecção é a parte mais importante do Snort [36]. Nela é feita a procura de intrusões nos pacotes utilizando suas regras. Estas regras são transformadas em estruturas de dados e são combinadas com todos os pacotes. Em caso de combinação, a ação definida na regra é acionada, caso contrário, o pacote é descartado.

A detecção de intrusão pode ser comprometida pelo seu tempo de execução. Dependendo do poder de processamento da máquina e da quantidade de regras, o tempo de resposta do programa pode variar. Se o tráfego for grande, pacotes podem ser perdidos e provavelmente não terá uma reposta em tempo real. Estes são os principais fatores que afetam o mecanismo de detecção [36]:

Número de regras Quanto maior o número de regras, mais processamento é necessário por pacote.

Poder de processamento da máquina Um computador com baixo poder computacional pode ser ineficiente ao analisar os pacotes.

Velocidade do barramento usado pela máquina Um computador com baixa velocidade no barramento pode perder pacotes e não responder em tempo real.

Carga na rede Uma quantidade grande de dados trafegando na rede aumenta a quantidade de processamento a ser realizada pelo IDS.

Além destes fatores, o sistema de detecção deve aplicar regras em diferentes partes de um pacote, adicionando mais complexidade ao problema. Estas partes podem ser [36]:

Tabela 2.1: Estrutura do cabeçalho das regras do Snort.

Action	**Protocol**	**Address**	**Port**	**Direction**	**Address**	**Port**

Cabeçalho IP Verificar campos como endereço de origem e destino, TTL, flags e etc.

Camada de transporte Procurar padrões em diferentes campos de diferentes protocolos como TPC, UPD ou ICMP.

Camada de aplicação Existem opções para alguns cabeçalhos da camada de transporte, mas geralmente isso é alcançado utilizando métodos alternativo como *offset* para o local onde está o dado que é procurado.

Carga do pacote Devem ser criadas regras que procuram por sequência de texto ou *bytes* dentro dos dados do pacote.

Logging and Alerting System

O sistema de alerta e registro é responsável por gerar alertas ou registrar atividades de acordo com as instruções descritas na regra, no caso de detecção realizada no motor de detecção [36].

2.2.2 Estrutura das Regras do Snort

As regras do Snort são simples de escrever e proveem uma estrutura poderosa para detectar uma grande variedade de atividades suspeitas na rede [38]. Uma regra no Snort pode ser dividida entre duas estruturas básicas: o cabeçalho e as opções.

Cabeçalho

O cabeçalho contém informações sobre qual ação deve ser tomada quando é feita uma detecção. Também são informados alguns critérios observados durante a avaliação das regras. O cabeçalho da regra é dividido em seis partes como demonstrado na Tabela 2.1.

Action: A parte *Action* determina quais ações serão tomadas quando um assinatura coincidir com o pacote. As principais ações tomadas são de alertar ou logar a mensagem da regra.

Protocol: A opção *Protocol* é usado para aplicar a regra a um protocolo específico. Por exemplo, se na regra o campo protocolo é preenchido com TCP, apenas os pacotes TCP serão inspecionados.

Tabela 2.2: Estrutura das opções das regras do Snort.

Opção 1	**Opção 2**	**Opção 3**
Palavra Chave 1 : Valor 1	Palavra Chave 2 : Valor 2	Palavra Chave 3 : Valor 3

Address: O campo *Address* define o endereço IP da fonte e destino. Pode ser preenchido um ou mais endereços ou uma sub rede. O campo *Direction* determina qual dos dois campos de endereço é a fonte ou destino. Por exemplo, se o campo *Direction* é ->"o endereço da esquerda é o da fonte.

Port: No caso do TCP ou UDP, a opção *Port* diz a porta de fonte e destino do pacote. Ela é afetada pelo campo *Direction* igualmente ao campo *Address*.

Direction: A parte *Direction* determina qual dos endereços e portas é o da fonte e destino.

Opções

As opções das regras formam o coração do motor de detecção de intrusão, combinando facilidade de uso com eficiência e flexibilidade [39]. Todas as opções são separadas por ponto e virgula (;) e as palavras chaves das opções são separadas por dois pontos (:). Esta estrutura pode ser visualizada na Tabela 2.2. A seguir temos um exemplo de regra onde as opções estão entre parênteses.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC
/etc/passwd"; flow:to_server,established; content:"/etc/passwd"; nocase;
http_uri; metadata:service http; classtype:attempted-recon; sid:1122;
rev:8;)
```

Estas opções serão detalhadas nas Tabela A.1, Tabela A.2, Tabela A.3 e Tabela A.4.

Existem quatro principais categorias de opções:

General Options São opções que proveem informações sobre a regra, mas não afeta durante a detecção [39]. A Tabela A.1 apresenta as oitos opções gerais do Snort.

Payload Detection Options Esta são as opções que procuram assinaturas dentro do *payload* do pacote e podem se inter-relacionar [39]. A tabela Tabela A.2 apresenta as opções de *payload* do Snort.

Nesse conjuto de opções está uma das principais opções, a opção *content*. Ela é responsável por definir o conteúdo específico que será procurado nos dados do pacote e ativar uma resposta baseada nestes dados. Sempre que o conteúdo da opção *content* é utilizada para o casamento de padrão, o algoritmo

de casamento de padrão Boyer-Moore é usada e testes são feitos na carga do pacote [39].

A opção *content* pode se tornar consideravelmente complexa, pois podem ser misturados texto e dados binários [39]. Os dados binários são encapsulados com o caracter *pipe* (|) e são representados em *bytecode*. *Bytecodes* representam dados binários em hexadecimal e é um método eficaz de representar dados complexos. A a seguir temos uma regra com a mistura de texto e *bytecode*.

```
alert tcp any nay -> any 139 (content:"|5c 00|P|00|I|00|P|00|E|00
5c|";)
```

Non-Payload Detection Options Estas são as opções que procuram por assinaturas nos cabeçalhos da camada de enlace, rede e transporte [39]. A Tabela A.3 apresentam as opções *non-payload* do Snort.

Post-Detection Options Esta opções determinam ações que devem ser tornadas após a detecção de uma vulnerabilidade [39]. A Tabela A.4 apresenta as opções de pós detecção do Snort.

2.2.3 Qualidade da Detecção

A qualidade do processo de detecção está atrelado a qualidade das regras que foram utilizadas. Existem várias regras que são especificas de alguns sistemas, como por exemplo a regra:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"SERVER-IIS Microsoft
ASP.NET viewstate DoS attempt"; sid:15959; gid:3; rev:5; classtype:attempted-dos;
reference:cve,2005-1665; reference:url,osvdb.org/show/osvdb/16195; metadata:
engine shared, soid 3|15959, service http, policy balanced-ips drop, policy
security-ips drop, policy max-detect-ips drop;)
```

Esta regra irá alertar qualquer pacote que possuir endereço IP de origem cadastrado na variável $EXTERNAL_NET (any por padrão) e endereço de IP destino da rede local conectando em uma porta HTTP. Note que esta regra gera um alerta para ataques de negação de serviço em servidores IIS [29], logo ela apenas deveria ser ativada quando se deseja proteger uma aplicação rodando em um servidor IIS.

Criar regras de qualidade é muito difícil, pois quanto mais especifica for a regra, maior a chance de achar uma vulnerabilidade real, porém maior a chance de deixa-la passar despercebida. Já quando temos regras com menos detalhes, a chande de encontrar o incidente é maior, porém a quantidade de falsos positivos aumenta consideravelmente.

2.3 Apache Storm

Storm [28] é um sistema para processamento de fluxo de dados em tempo real de maneira distribuída, confiável e tolerante a falhas. O trabalho é dividido entre diferentes tipos de componentes e cada um responsável por uma tarefa simples e específica.

2.3.1 Componentes do Storm

Um *cluster* Storm segue o modelo "mestre-escravo" onde os processos do mestre e dos escravos são coordenados pelo *ZooKeeper*. Logo, o Storm tem três componentes principais [21]:

Nimbus: Este é o nó mestre em um *cluster* Storm. Ele é responsável por distribuir o código da aplicação entre os vários nós escravos, atribuindo tarefas para diferente máquinas, monitorando tarefas por qualquer falha e as re-setando quando é necessário [21].

O *Nimbus* não possui estado e guarda todos os seus dados no *ZooKeeper*. Há apenas um nó Nimbus em um *cluster* Storm. Ele é projetado para ser *fail-fast*, então quando o *Nimbus* falha, ele pode ser reiniciado sem afetar as tarefas que estão rodando nos nós trabalhadores.

Supervisor nodes: Estes são os nós trabalhadores no Storm *cluster*. Cada nó supervisor roda um supervisor *daemon* que é responsável por criar, iniciar e parar processos do trabalhador para executar tarefas atribuídas àquele nó. Como o *Nimbus*, um supervisor *daemon* também é *fail-fast* e guarda todo seu estado no *ZooKeeper*, de forma que ele pode ser reiniciado sem perder o estado atual. Um único supervisor *daemon* pode controlar múltiplos processos trabalhadores rodando em uma máquina [21].

Cada nó trabalhador roda uma Java virtual machine (JVM), onde podem rodar um ou mais *executors* ou executores. Executores são compostos de uma ou mais *tasks* ou tarefas onde o processamento em um *bolt* ou *spout* é realizado. Portanto, tarefas proveem paralelismo "intra-bolt/intra-spout", executores proveem paralelismo "intra-topologia", e os processos trabalhadores servem como um *container* na máquina que executará uma topologia do Storm [4].

ZooKeeper: É uma aplicação que, de maneira confiável, coordena e proveem serviços de comunicação entre os processos do sistema distribuído. Como o Storm é um sistema distribuído, ele também usa o *ZooKeeper* para gerenciar vários processos. Todos os

estados associados com o *cluster* e as várias tarefas enviadas ao Storm são armazenadas no *ZooKeeper*. O *Nimbus* e seus nós supervisores não se comunicam entre si, mas através do *ZooKeeper*. Todos os dados são armazenados em um *Zookeeper* e ambos, *Nimbus* e *supervisor daemons*, podem ser finalizados de maneira inesperada sem afetar o funcionamento do *cluster*.

A Figura 2.4 ilustra a arquitetura de um cluster Storm:

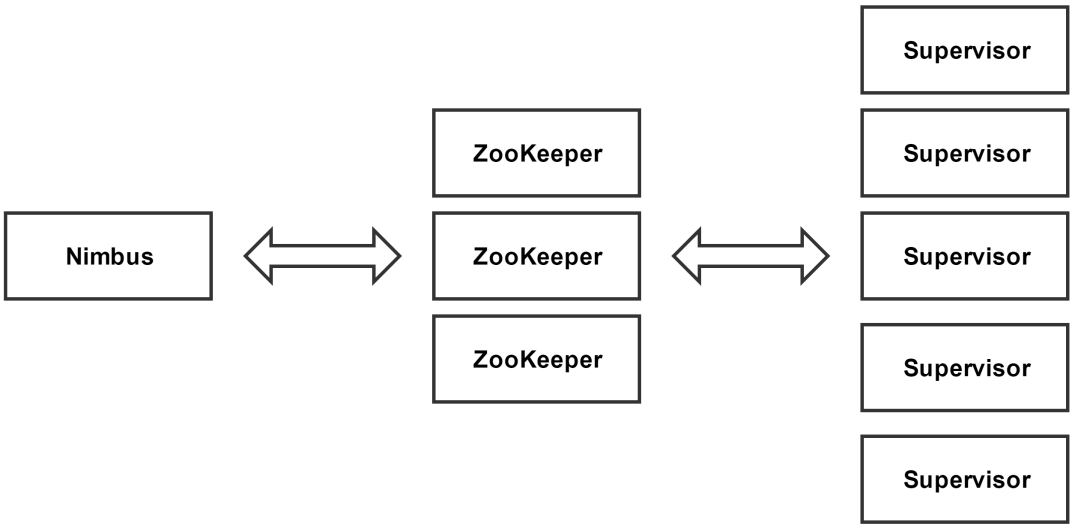

Figura 2.4: Arquitetura de um cluster Storm [21].

2.3.2 Modelo de dados no Storm

A unidade principal de dados que pode ser processada pelo Storm é a *tuple* ou tupla que é uma lista de valores nomeados (pares do tipo chave-valor)[16]. Cada tupla consiste de uma lista predefinida de campos. O valor de cada campo pode ser *byte*, *char*, *interger*, *long*, *float*, *double*, *boolean* ou vetor de *bytes* [21]. O Storm também prover uma API para definir tipos de dados customizáveis, que podem ser convertidos em campos de tuplas.

A tupla é tipificada dinamicamente, ou seja, apenas os nomes das tuplas devem ser definidos e não seu tipo. A escolha de tipificar a tupla dinamicamente ajuda a simplificar a API e torna o sistema mais fácil de usar. Além do mais, como a unidade de processamento no Storm pode processar múltiplos tipos de tuplas, é muito mais prático não declarar tipos de campos.

2.3.3 Definição de uma topologia

No Storm, a topologia é uma abstração que define um grafo na computação que será realizada [21]. Para se processar dados no Storm, deve-se criar uma topologia e implantá-la em um *cluster* Storm. Uma topologia pode ser representada por um grafo acíclico dirigido, onde cada nó faz um tipo de processamento e passa para frente o resultado para o próximo fluxo de nós. A Figura 2.5 representa um exemplo de topologia no Storm onde a entrada de fluxo de dados é tratada pelo *spout*, que atua como um adaptador que conecta a topologia a uma fonte de dados. O *spout* passa os dados para o componente chamado *bolt*, que transforma ou processa estes dados. Um *bolt* pode salvar este dado em algum sistema de armazenamento ou passar para outro *bolt* processá-lo. Uma topologia pode ser imaginada como uma cadeia de *bolts* na qual cada um faz um processamento ou transformação nos dados entregados pelo *spout* [7]. A seguir, temos mais detalhes sobre os componentes da topologia:

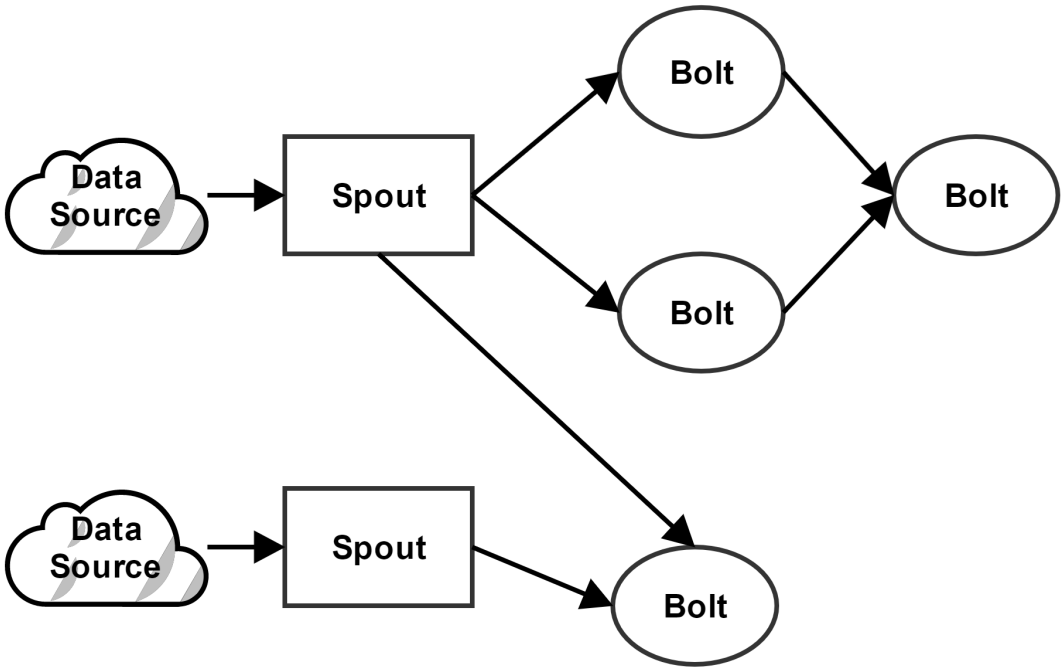

Figura 2.5: Exemplo de topologia no Storm [16].

Stream: De acordo com Jain e Nalya [21], esta pode ser considerada a abstração chave do Storm. Um *stream* é uma sequência ilimitada de tuplas que pode ser processadas em paralelo pelo Storm. Cada *stream* pode ser processado por um ou mais *bolts*. Além do mais, o Storm pode ser visto como uma plataforma que transforma *streams*. Na Figura 2.5, o *stream* é representado por setas sinalizando o fluxo de dados sendo transmitidos entre *bolts* e *spouts*.

Spout: O *spout* é a fonte de tuplas em uma topologia Storm. Ele atua como um adaptador que conecta a topologia a uma fonte de dados, transformando-os em tuplas e as emitindo como *stream* [16]. Portanto, ele é responsável por ler e escutar os dados de uma fonte externa e alimentar a topologia com informações. Ele é capaz de, por exemplo, escutar uma fila de mensagens e emitir todos os dados lidos para um *stream*.

O *Spout* possui quatro métodos importantes [21]:

nextTuple(): Este método é invocado pelo Storm para recuperar a próxima tupla da fonte de dados.

ack(): Este método é chamado pelo Storm quando uma tupla com um determinado ID tem seu processamento finalizado. Ele é útil para realizar pós processamento nas mensagens computadas, como marcá-las como processadas e retirar a mensagem da fila para que elas não sejam lidas novamente.

fail(): Este método é chamado pelo Storm quando ele identifica que a tupla com um ID específico não foi processada com sucesso ou não foi processada no intervalo de tempo definido nas configurações.

open(): Este método é invocado quando o *Spout* é inicializado. Quando o *Spout* deve ser conectar com uma fonte de dados externa, a lógica de conexão deve ser implementada neste método e a busca dos dados deve ser feita no método *nextTuple()* para poder enviar estes dados para os outros nós do *cluster*.

Bolt: O *bolt* é unidade principal de processamento na topologia do Storm e é responsável por transformar o *stream*. Ele também pode ser pensando com o operador ou as funções para a computação desejada [16]. O *bolt* pode receber como entrada qualquer número de *streams*, processar os dados e opcionalmente gerar um ou mais *streams*.

Em um cenário ideal, cada *bolt* deve ser responsável por fazer uma transformação simples das tuplas, e vários *bolts* sincronizados fazendo transformações complexas [21]. O *Bolt* possui dois métodos principais [21]:

execute(): Este método é executado por cada tupla que chega pelos fluxos de dados ligado ao *Bolt*. Este método pode fazer qualquer processamento que for implementado na tupla e então produzir uma saída, podendo enviar mais tuplas para outros *Bolts* ou salvando os resultados em um banco de dados.

prepare(): Um *Bolt* pode ser executado por múltiplos processos trabalhadores numa topologia do Storm. A instância de um *Bolt* é criada na máquina cliente e depois serializada e enviada ao *Nimbus*. Quando o *Nimbus* cria instâncias de processos trabalhadores para a topologia, ele envia o *Bolt* serializado para estes trabalhadores. Após este processo o método *prepare* é chamado. Nele, o *Bolt* deve ser configurado para iniciar o processamento de tuplas. Qualquer estado pode ser mantido usando varáveis de instância que podem ser serializadas e deserializadas.

2.4 Apache Kafka

O Apache Kafka[12] é um *publish-subcribe messaging system*, ou seja, ele suporta a publicação de mensagens para um tópico de mensagens específico e os assinantes podem receber estas mensagens assinando o tópico. O Kafka é capaz de conectar vários aplicativos em tempo real fazendo o roteamento de mensagens usando o mecanismo de publicação de mensagens [15]. Ele também pode prover integração entre as informações dos produtores e consumidores sem bloquear os produtores e sem permitir que saibam quem são os consumidores finais. O Kafka possui as seguintes principais características [15]:

Envio de mensangens persistente: Nenhum tipo de informação deve ser perdida para agregar valor real a uma aplicação de *big data*. Por isso o Kafka é projetado com estruturas de arquivos com complexidade O(1) que prover performance em tempo constante mesmo em volumes muito grandes de mensagens armazenadas que estão na ordem de *terabytes*. Com o Kafka, as mensagens são armazenadas em disco e também replicadas no *cluster* para evitar perda de dados.

Alta vazão: O Kafka é projetado para trabalhar com *hardware* comum e lidar com cetenas de *megabytes* de leitura e escrita por segundo, com um número muito alto de clientes.

Distribuído: O Apache Kafka é projetado para suportar *clusters* e por isso suporta particionamento de mensagens por todos os servidores Kafka e consumo distribuído pelo *cluster* de máquinas consumidoras.

Suporte a múltiplos clientes: O Kafka suporta integração trivial a clientes de plataformas diferentes como Java, .NET, PHP, Ruby e Python.

Processamento em tempo real: As mensagens produzidas pelas *threads* produtoras devem ser visíveis às *threads* consumidoras imediatamente.

A Figura 2.6 ilustra uma aplicação de *big data* típica usando o Kafka como sistema de envio de mensagem. Diferentes produtores podem enviar mensagens para diferentes agentes do Kafka e eles não têm nenhuma ligação direta com os produtores.

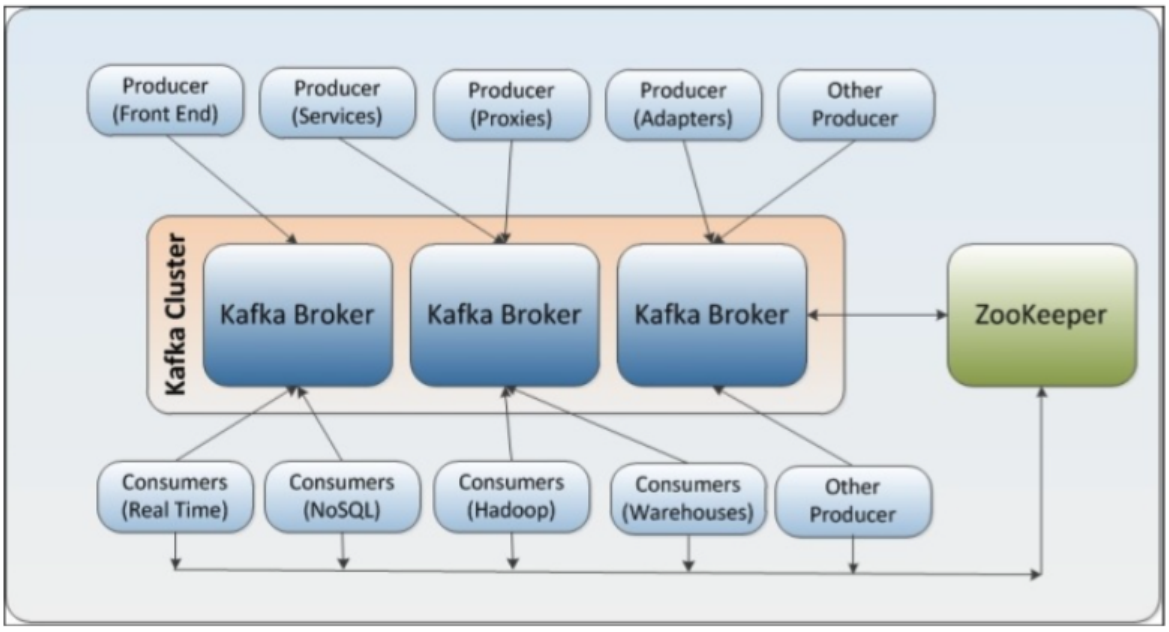

Figura 2.6: Diagrama de uma arquitetura típica utilizando o Kafka adaptado de [15].

2.5 Cassandra

Cassandra é um sistema de armazenamento de dados de código aberto. Ele foi criado inicialmente pelo Facebook[9] em 2007 para solucionar o problema de busca na caixa de entrada dos usuários, onde era necessário manusear um volume muito grande de dados [19]. A diferença do Cassandra para os outros bancos de dados tradicionais é seu potencial para ser distribuído e gerenciar quantidades consideravelmente grandes de dados estruturados por meio de vários servidores com *hardware* comum enquanto prover alta taxa de disponibilidade sem nenhum ponto de falha [26].

2.5.1 Modelo de dados do Cassandra

O Cassandra possui três *containers* aninhados [33]. O da camada mais superior é o *keyspace*. O *keyspace* pode ser comparada com uma *database* em um Sistema Gerenciador de Banco de Dados Relacional (SGBDR). Dentro do *keyspace* encontramos as tabelas, que são uma versão das tabelas de um SGBDR, porém mais flexível. Uma tabela é basicamente um mapeamento ordernado de mapeamentos ordenados [33]. Cada tabela possui uma chave primária que é chamada de *row key* ou *partition key*. Dentro da tabela há as partições que são associadas com um conjunto de células. Cada célula tem um nome e um valor. Elas podem ser comparadas com as colunas de um banco de dados tradicional e a partição com uma linha. A Figura 2.7 ilustra a organização do modelo de dados no Cassandra.

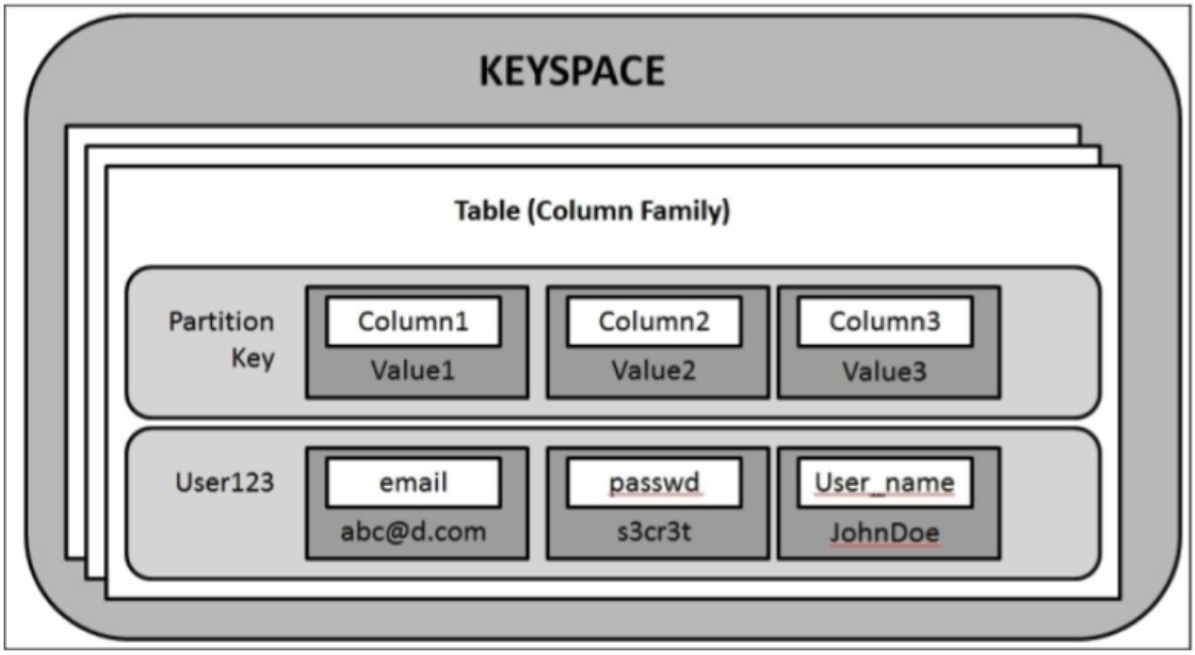

Figura 2.7: Modelo de dados no Cassandra [33].

2.5.2 Eficiência do Cassandra

Em um estudo realizado por Rabl et al. [10], o Cassandra superou vários bancos de dados populares em termos de *performance*. Entre eles estavam HBase[14], Voldemort[44], Redis[40], VoltDB[45] e MySQL[32]. A Figura 2.8 mostra o desempenho em um dos teste onde foi realizado somente escrita e aumentando o número de nós.

Note que o resultado para um nó é bem similar para todos os bancos de dados, porém o Cassandra possui um desempenho escalar sendo quase linear com o número de nós.

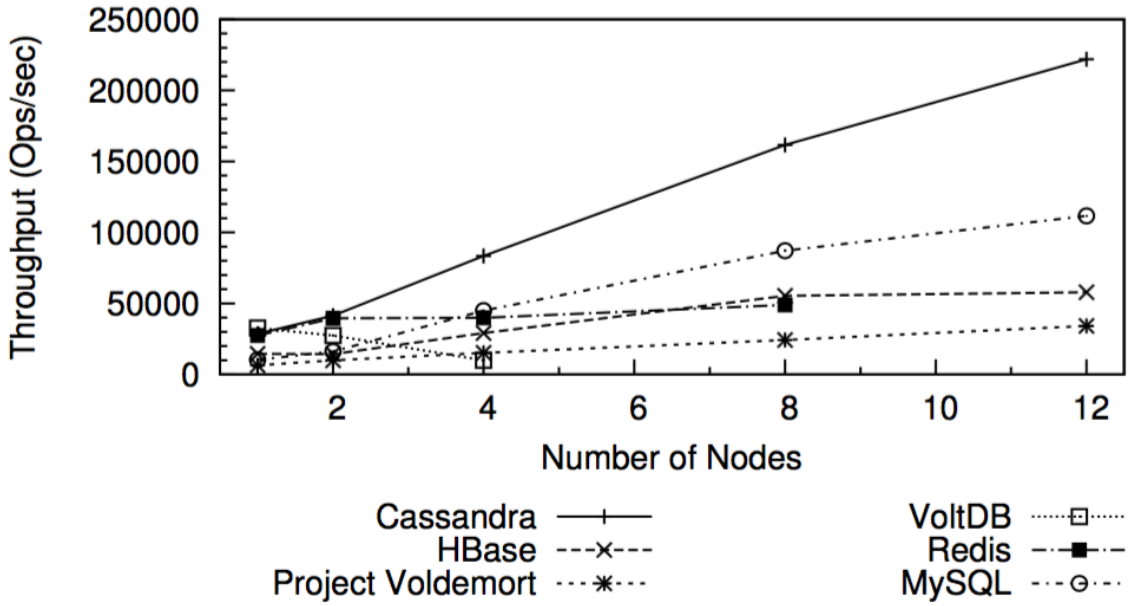

Figura 2.8: Desempenho dos bancos de dados Cassandra, HBase, Voldemort, Redis, VoltDB e MySQL na escrita de dados. [33].

2.6 Resource Monitor e Storm Engine

Resource Monitor e o *Storm Engine* são componentes de um sistema de monitoramento de ambientes computacionais distribuídos em larga escala e em tempo real desenvolvido por [43]. Este sistema é capaz de coletar e analisar recursos dos ativos que deseja-se monitorar e se baseia em uma arquitetura de troca de mensagens entre o ambiente computacional monitorado e o sistema de monitoramento, através de uma conexão direta entre os computadores ou por meio de um fila de mensagens. O Resource Monitor é o software responsável por coletar os dados das máquinas monitorada e o Storm Engine é o sistema de monitoramento que irá analisar os dados coletados de maneira distribuída usando o Apache Storm.

Estes programas possuem duas abordagens para a distribuição dos dados. A primeira é sem fila de mensagem usando o *Netty* [35] (Figura 2.9) para trocar mensagens entre os clientes e o servidor. Esta arquitetura foi projetada para monitorar ativos que não necessitam transmitir uma larga quantidade de dados e sua grande vantagem é o suporte a criptografia (SSL/TLS - StartTLS) no canal de comunicação [43].

Já a segunda abordagem utiliza o *Kafka*, como fila de mensagem, para realizar a troca de mensagens entre o *Resource Monitor* e o *Storm Engine* Figura 2.10. A arquitetura utilizando o Kafka é recomendada para monitorar ativos que necessitam transmitir um

grande volume de dados, pois apesar de possui uma velocidade inferior em relação ao *Netty*, ela suporta um quantidade muito maior de tráfego [43]. Outra desvantagem é não poder cifrar o canal.

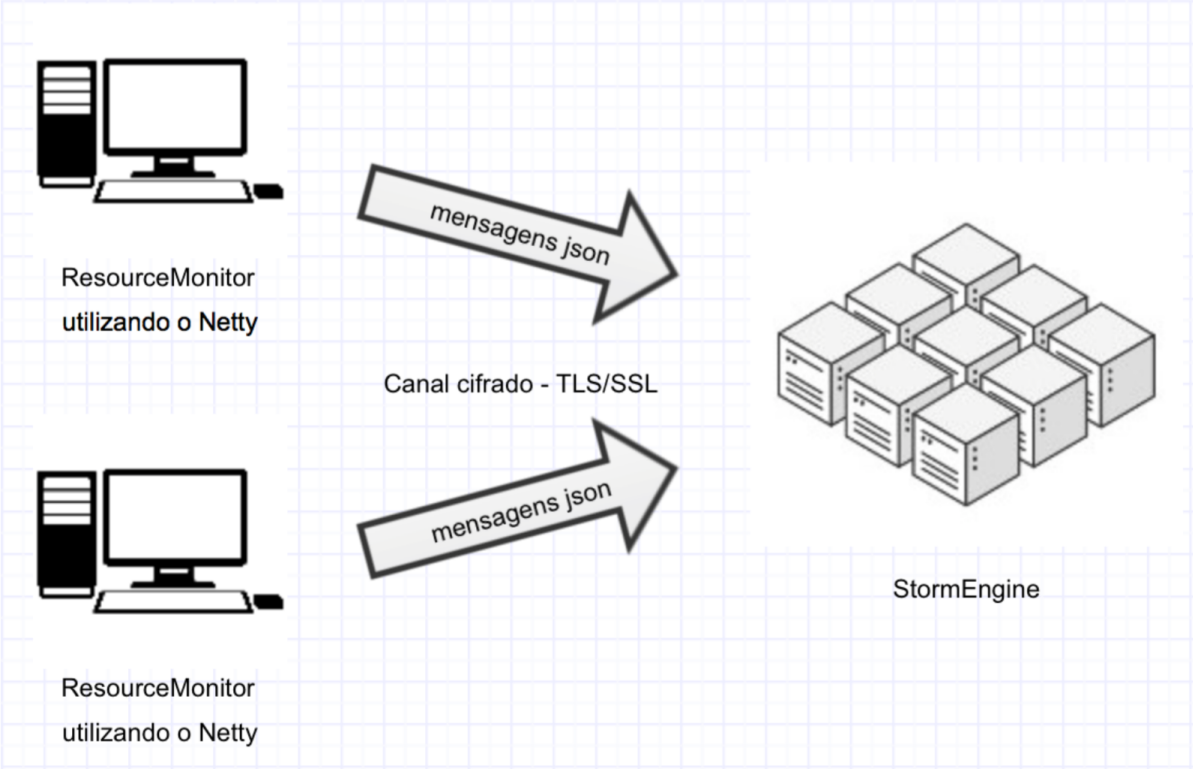

Figura 2.9: Arquitetura sem fila de mensagem em [43].

2.7 Síntese do capitulo

Neste capítulo vimos a que IDS é um sistema capaz de detectar atividades ilícitas em *software* de terceiros ou redes de computadores. Além do mais, sistemas detectores de intrusão podem ser classificado em duas categorias principais: Os HIDS que detectam ataques processando informações a nível de aplicação e os NIDS que detectam intrusões processando informações da rede. Em seguida, foi apresentado os três componentes de um IDS. A fonte de dados, responsável por alimentar o sistema com dados que serão referencias para o IDS achar vulnerabilidades , o motor de análise, encarregado por analisar os dados da fonte de dados e identificar ataques, e o gerenciador de resposta que é responsável por armazenar e informar ao usuário que um ataque foi identificado. Logo após, é apresentado onde o IDS é geralmente posicionado e os impactos que esse posicionamento pode gerar na rede.

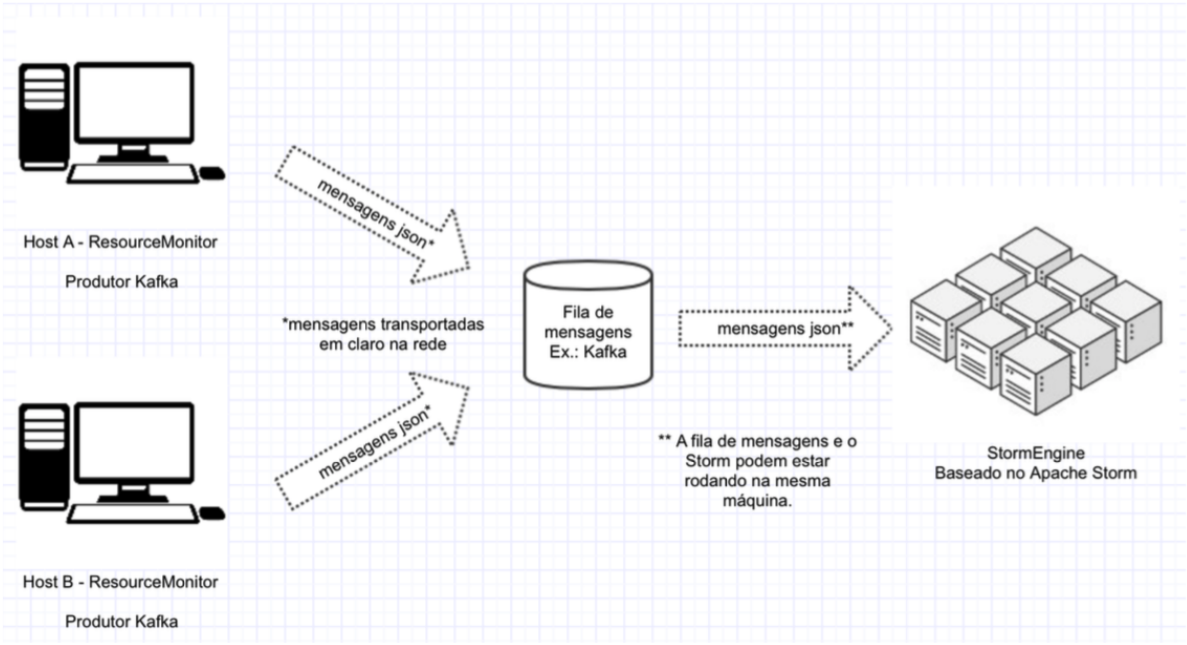

Figura 2.10: Arquitetura com fila de mensagem em [43].

Na seção seguinte, foi apresentando o Snort, um sistema detector de intrusão livre, baseados em regras, as quais são aplicados nos pacotes que ele captura. Explicou-se com detalhes sua arquitetura e a estrutura de suas regras. A descrição do Snort é importante para este projeto pois, esta solução utiliza muitos elementos deste IDS.

Logo após, é apresentado o Apache Storm, *framework* de processamento de fluxo de dados em tempo real. O Storm é capaz de distribuir o processamento de dados em vários nós de um *cluster* através da abstração da topologia. Esta que é composta do *Stream* que é uma sequencia de dados que serão processados, do *Spout* que alimenta a topologia com dados de uma fonte externa e os transforma em *stream*, e o *Bolt* que processa os *streams*.

Na próxima seção, temos o Apache Kafka que é uma fila de mensagem capaz de conectar vários aplicativos em tempo real. Sua grande vantagem é sua capacidade de distribuir, aumentando a escalabilidade deste projeto.

Logo após, foi apresentado o Cassandra, um banco de dados comumente utilizado em aplicação com grande volume de dados. Ele possui potencial para ser distribuído, que da mesma maneira do Kafka, aumenta a escalabilidade deste projeto.

Por ultimo, temos o sistema desenvolvido por [43], que é um sistema para monitoramento de ambientes computacionais distribuídos. Este projeto irá servir como ponto de partida para o desenvolvimento do Storm IDS.

A seguir, será discutido a arquitetura e os módulos que compõem esta solução. Todos as ferramenta e conceitos descritos anteriormente foram utilizados para construir o sistema

detector de intrusão distribuído proposto no próximo capítulo.

Capítulo 3

Projeto e Implementação

A solução desenvolvida neste projeto é um desdobramento de [43], um sistema escalável para processamento de dados distribuído e em tempo real, onde este foi adaptado para criar uma solução de detecção de intrusão para ambientes distribuídos. Por ser um projeto de um IDS baseado no Storm, ele foi chamado de *Storm IDS*.

Como visto anteriormente na seção 2.6, o sistema desenvolvido por [43] possui duas arquiteturas, uma com comunicação direta entre os clientes e servidor e outra usando fila de mensagem. Como a solução utilizando o Kafka é mais eficiente para ambiente com alta taxa de transmissão de dados, esta abordagem foi escolhida para o desenvolvimento deste projeto.

Para alcançar o objetivo, foi criado um *plugin* para o *Resource Monitor* que é capaz de capturar pacotes, processá-los e enviar para o *Storm Engine*. No *Storm Engine*, outro *plugin* foi criado para procurar vulnerabilidades e identificar se está ocorrendo algum ataque na rede. Como o volume trafegado pela rede pode se tornar muito alto, foi escolhida a arquitetura com fila de mensagem para este novo projeto, pois ela suporta uma quantidade muito maior de dados do que a sem fila de mensagens e assim evita-se que pacotes sejam perdidos.

Durante os primeiros testes foram identificados alguns gargalos que prejudicavam a escalabilidade do sistema. Os dados eram enviados no formato JSON e durante o processo de detecção de intrusão eram necessárias várias conversões. Então foram realizadas várias modificações nos dois componentes para que os dados sejam enviados como sequencia de *bytes*, evitando conversões desnecessárias.

Na primeira seção deste capítulo serão apresentados mais detalhes sobre a arquitetura do sistema implementado.

Nas outras seções serão detalhados os cinco módulos que compõem o sistema de detecção de intrusão proposto. A Figura 3.1 ilustra estes módulos, demostrando como eles se relacionam. A seguir, há uma breve descrição dos cinco componentes do Storm IDS:

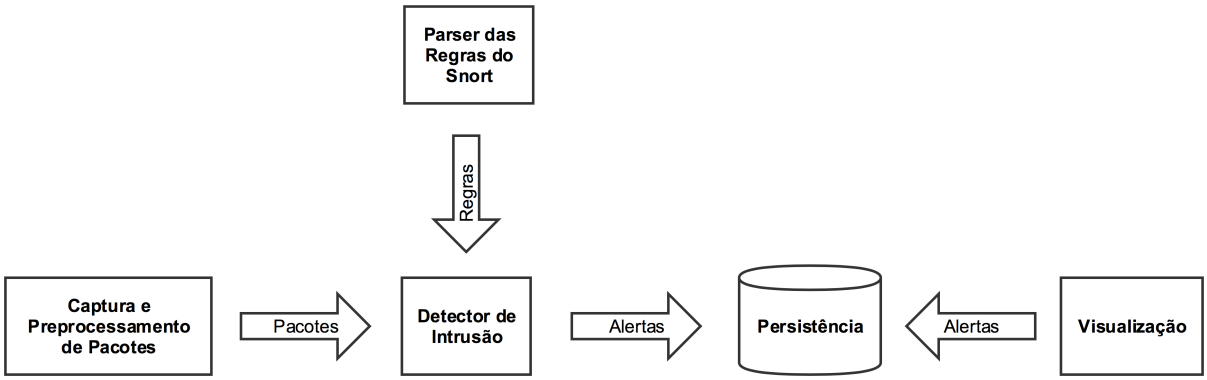

Figura 3.1: Módulos do Storm IDS.

Captura e Preprocessamento de Pacotes: Responsável por capturar os pacotes da rede, separar apenas as informações úteis para a detecção e enviar para o *Storm Engine*, localizado no *Resource Monitor*.

Parser das Regras do Snort: Responsável por ler os arquivos com as regras do Snort e criar estruturas de dados que facilitarão a busca por ataques.

Detector de Intrusão: Este é o coração do IDS. Aqui as regras são casadas com os pacotes e ataques são identificados

Persistência: Responsável por salvar os dados de alertas gerados pela detecção de intrusões.

Visualização dos Resultado: Este componente permite ao usuário visualizar os logs gerados pelo Storm IDS.

3.1 Arquitetura do Storm IDS

A solução criada utiliza os programas *Resource Monitor* e *Storm Engine* para distribuir os dados a serem processados. O *Resource Monitor* deve se executado nas máquinas que serão monitoradas e o tráfego capturado deve ser enviado para a fila de mensagens implementada pelo Kafka que será consumida pelo *Storm Engine*. Quando algum alerta é gerado no mecanismo de detecção, o *Storm Engine* insere esses dados em um banco de dados. Para este projeto foi utilizado o Cassandra. A Figura 3.2 ilustra esta arquitetura.

O *Resource Monitor* foi criado dando suporte a *plugins* que são capazes de capturar diferentes tipos de dados de diferente fontes. Por exemplo, nos casos de usos apresentados em [43], o uso do processador, uso da memória e uso do sistema de arquivo são monitorados no cliente. Estes *plugins* enviam dados para o *Storm Engine* com uma determinada

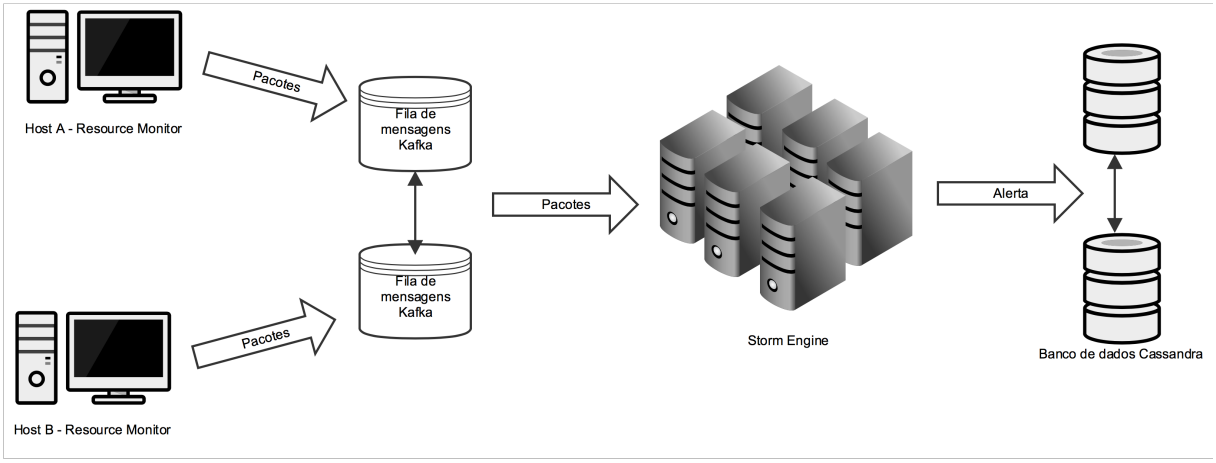

Figura 3.2: Arquitetura da nova solução adaptada de [43].

frequência. Para o Storm IDS foi criado o *plugin PacketCapture* que irá enviar todos os pacotes capturados a cada um segundo. Este precedimento será explicado com mais detalhes na próxima seção.

No caso do *Storm Engine*, sua estrutura é baseada nas topologias do Storm. A topologia *Kafka Topology*, ilustrada na Figura 3.3 [43], foi adaptada para suportar detecção de intrusão. O *Kafka Spout* recebe os dados vindos dos computadores que estão rodando o *Resource Monitor* e passa essa informação para o *Multiplexer Bolt*. Ele irá identificar quais os tipos dos dados que estão sendo enviados e irá enviar e distribuir para os *bolts* que são capazes de processá-los. Para a detecção de intrusão, foram criado dois *bolts*, o *Network Data Bolt*, que irá fazer o *parser* das regras quando for inicializado e casar estas regras com os pacotes, e o *Log Matches* que irá receber os dados da detecção e salvar no banco de dados. A Figura 3.4 ilustra a adaptação da *Kafka Topology* para implementar o IDS.

3.1.1 Justificativa da arquitetura

Como mencionado no capítulo anterior, a detecção de intrusões na rede é um processo muito custoso e são vários os motivos que podem afetar o desempenho de um IDS [36]. Em um cenário onde o número de regras e de pacotes se torna muito alto, arquiteturas como a do Snort não conseguirão escalar e se tornarão um gargalo na rede [11]. A primeira vantagem do Storm IDS é seu potencial para o processamento de dados em larga escala, pois roda sobre *frameworks* conhecidos e comumente utilizados para aplicações de *big data* (Storm e Kafka).

No caso de uma rede onde existem várias entradas para a internet ou redes externa e em casos onde deve-se evitar, além das ameaças externas, os ataques internos também,

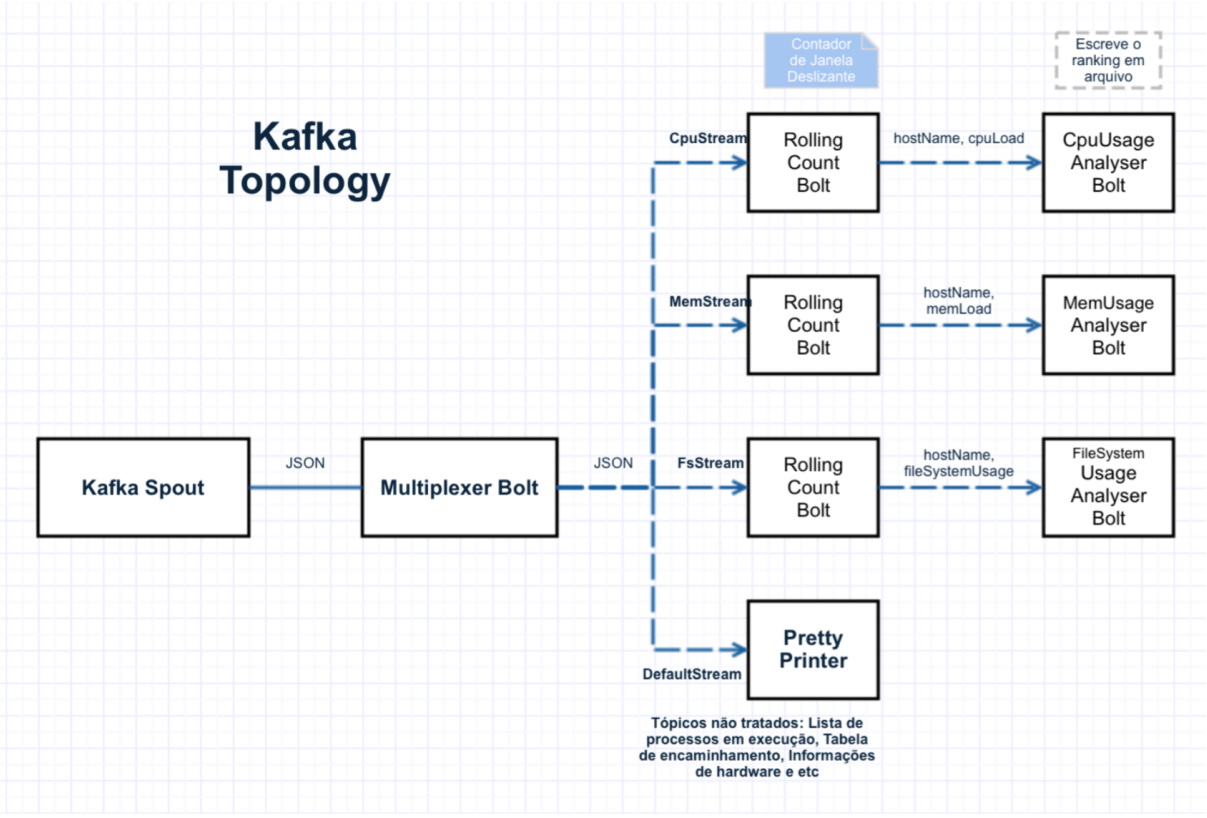

Figura 3.3: *Kafka Topology* original de [43].

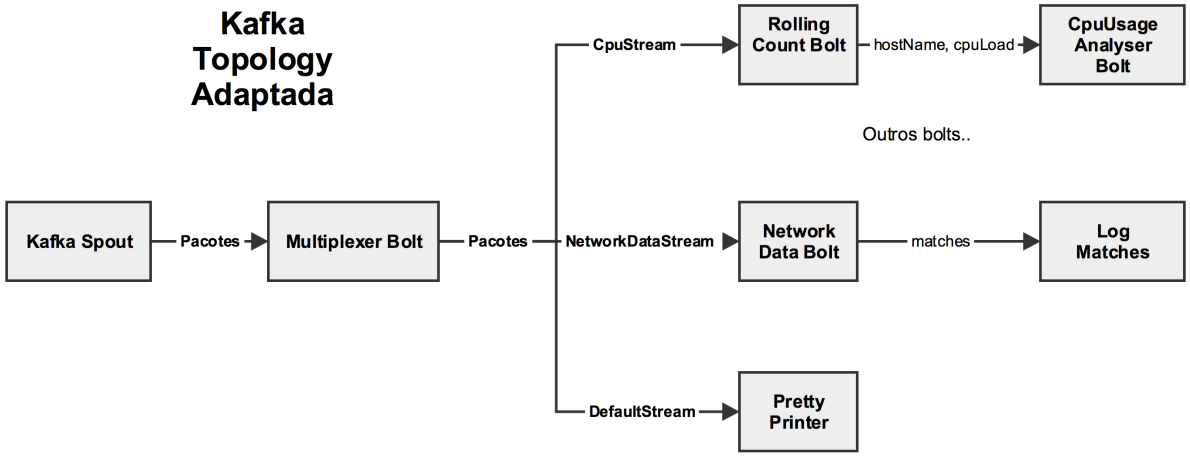

Figura 3.4: *Kafka Topology* da nova solução adaptada de [43] para implementar o IDS.

os NIDSs devem ser posicionados em vários locais estratégicos tornando praticamente inviável adotar esta solução. No Storm IDS, os dados são capturados em cada dispositivo que roda o *Resource Monitor* e isso evita a necessidade de uma fonte de captura de pacotes em cada nó da rede.

3.2 Captura e Preprocessamento de Pacotes

O primeiro passo do sistema detector de intrusão é a captura do tráfego. Neste projeto, a captura é feita em cada máquina monitorada por meio do *Resource Monitor*. Foi criado um *buffer* que utiliza a biblioteca pcap4j [46] para a captura do tráfego que é consumido pelo *PacketCapture plugin*. Também foi criado um arquivo de configuração para deixar o programa mais flexível. Nele pode ser configurado o endereço do *Zookeeper* do Kafka, dos *Brokers* do Kafka e um filtro de pacotes do tipo *Berkeley Packet Filter (BPF)*.

Ao iniciar o *plugin* é criada uma *thread* que irá capturar todos os pacotes que passarem pelo computador monitorado. Este processamento é feito pela classe *PacketCapture* que cria um *listener* que ficará escutando a interface da rede selecionada. Cada pacote capturado é preprocessado e transformado em um objeto do tipo *PacketData*, guardando apenas as informações que podem ser utilizadas no mecanismo de detecção de intrusão. O método principal da classe *PacketCapture* é o *getParcketData(Packet packet)* que é responsável por ler as informações do pacote e salvá-las em um objeto do tipo *PacketData*. Este método é capaz de identificar se o pacote faz parte dos protocolos IPv4 ou IPv6 e do protocolos TCP, UDP ou ICMP.

O objeto resultante deste processamento é alocado em uma fila de mensagem implementada pela classe *PacketQueue*. Esta classe possui dois métodos principais, o *addPacket(PacketData p)* que é usado para se adicionar pacotes na fila e o *getPackets()* que retorna todos os pacotes presente na fila para o *PacketCapturePlugin*, tornando a fila vazia para que novos pacotes sejam alocados.

A Figura 3.5 ilustra o processo de captura de pacotes.

3.3 Parser das Regras do Snort

Para a realizar a detecção de intrusões, é necessário uma base de assinaturas de vulnerabilidades. Para conseguir esses dados, é feito o *parsing* nas regras do Snort. O Snort disponibiliza esses dados no seu site gratuitamente em vários arquivos. Este módulo irá ler estes arquivos e transformar os dados em estruturas de dados que poderão ser usadas no mecanismo de detecção de intrusão. A classe *Rules* possui o método *get()* que retorna uma lista com todas as assinaturas tratadas após o *parsing* das regras. A assinaturas são representada pela classe *SnortSignature* que possui quatro atributos que representam diretamente a estrutura de uma regra conforme visto na seção 2.2.2:

Header: Cabeçalho da regra [39].

GeneralOptions: Opções que não influenciam na detecção e apresentam informações sobre a regra [39].

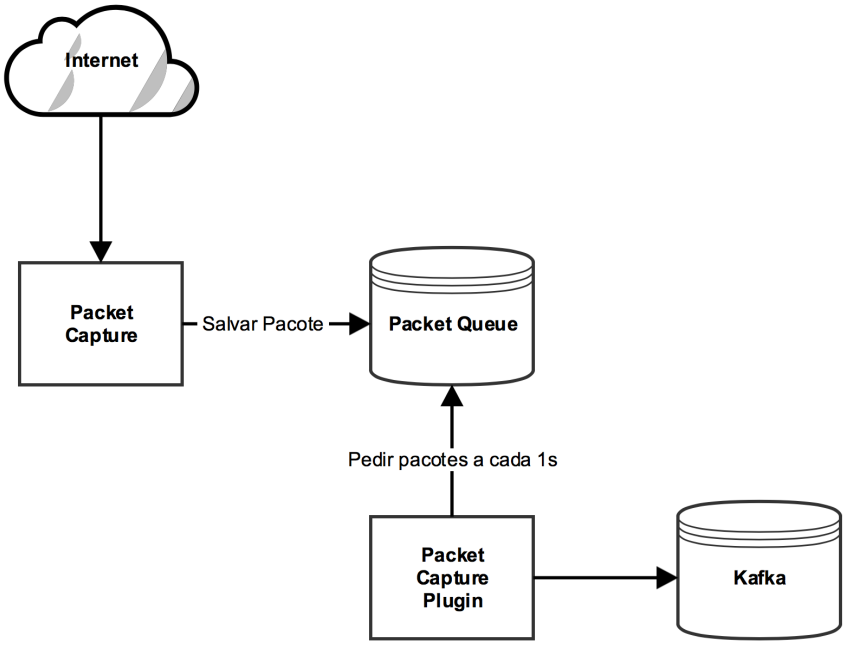

Figura 3.5: Captura de pacotes no Storm IDS.

NonPayloadOptions: Opções que atuam nos dados fora do *payload* do pacote [39].

PayloadOptions: Opções que atuam nos dados dentro do *payload* do pacote [39].

A Figura 3.6 ilustra a estrutura de uma assinatura no sistema. Note que as opções de pós detecção não serão tratadas pois elas são muito especificas do software do Snort e não estão no escopo deste projeto.

Ao iniciar o *parser* das regras, a assinatura é quebrada em duas partes: o cabeçalho e as opções. Como em todas as regras os cabeçalhos são separados das opções por parênteses, como descrito na Figura **??**, essa separação é facilmente realizada. Então é feito o *parsing* de todos os campos do cabeçalho em um objeto da classe *Header*. Já as opções são separadas por ponto e virgula (;). Isso também facilitada sua separação no texto da regra e para cada opção é verificado o seu tipo. Itera-se na lista de opções tratando cada uma e alocando em seu respectivo tipo.

3.3.1 Opção Content

A opção *content* é a mais importante e também mais complexa entre as opções presentes nas regras do Snort. Ela é responsável por fazer a detecção em profundidade no pacote. Ela se torna complexa por ter outras opções que podem modificar seu comportamento e o fato dela poder ser combinada com outras opções do mesmo tipo *content* na mesma

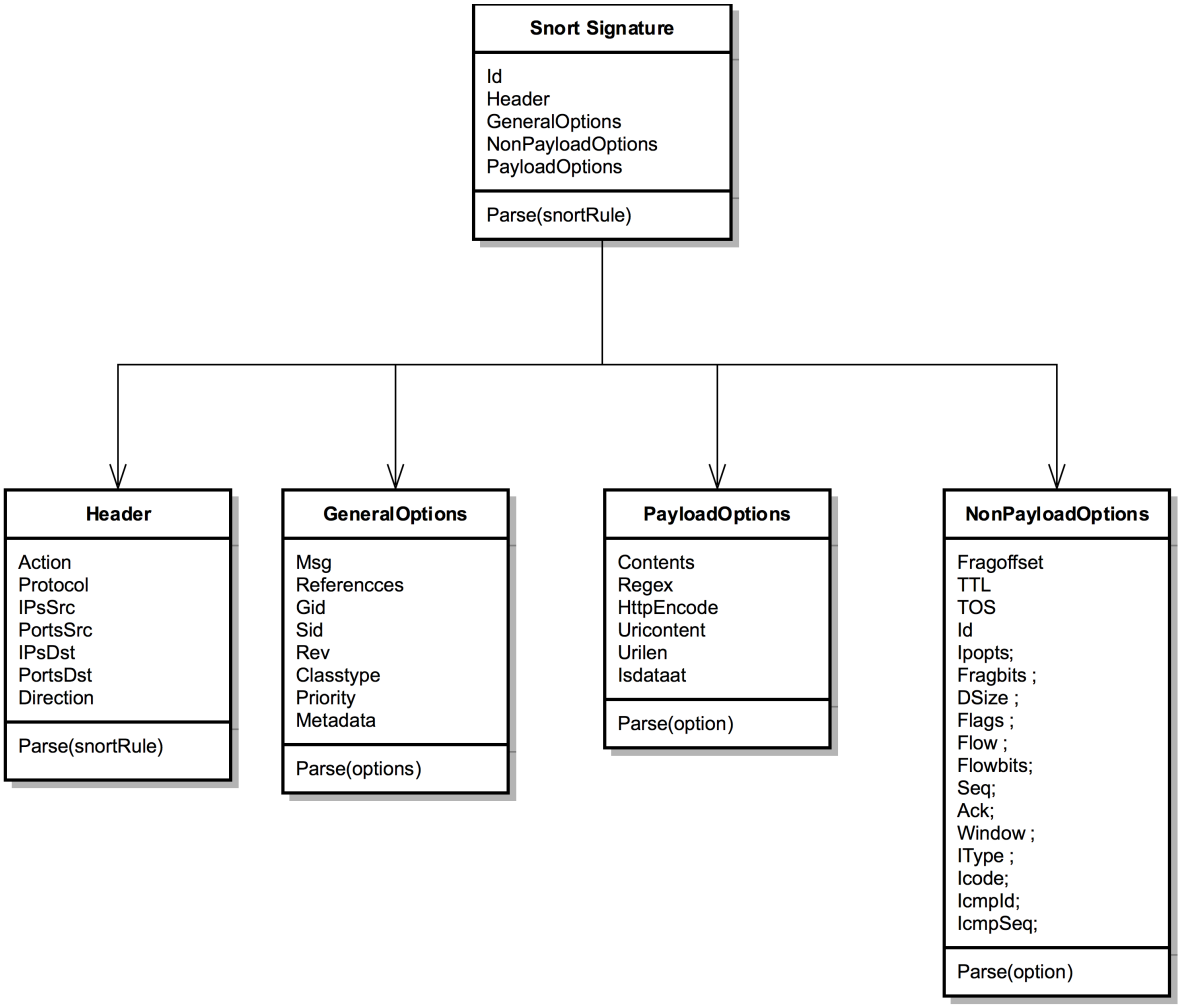

Figura 3.6: Diagrama de classes da estrutura de dados que representa as regras do Snort.

regra. Nela também pode ser escrito texto e código hexadecimal juntos. Para facilitar seu parsing foi criada a classe *Content* que representa uma única opção *content*.

A classe *Content* possui quatro métodos para realizar o parse no campo *content* de uma regra. O primeiro método é o *void parseContent(String data)* que recebe os dados da opção e realiza o parse nesses dados. Primeiro são removidos aspas (") e ponto e virgula (;) da string. Como explicado no capítulo 2, na opção *content* os dados representados por texto são separados dos binários por um *pipe* (|). Logo, é realizada uma iteração sobre todos os caracteres da string enviada ao método.

Quando um *pipe* é encontrado, o método *appendContent()* é chamado. Este método verifica se o dado é hexa decimal ou texto e dependendo do resultado chama o método *appendString()* (no caso que for texto) ou o método *appendHex()* (no caso que for hexa decimal). Estes métodos convertem ambos os dados de seu *encoding* para ISO 88591 por este ser capaz de representar todos os hexadecimais de 00 a FF. Isso facilitará a detecção

de intrusão usando expressões regulares.

3.3.2 Justificativa para o uso das regras do Snort

O Snort é uma das ferramentas mais conhecidas e mais utilizadas para detectar intrusões na rede. O fator principal para a eficiência da detecção de intrusão é a qualidade das regras. O primeiro motivo para a utilização das assinaturas do Snort neste projeto é fato da equipe responsável pelo Snort disponibilizá-las gratuitamente. Outro motivo é a reputação dessas regras, são bem aceitas pela comunidade e usadas como referência para outros IDSs. Por último, as regras seguem uma estrutura que pode ser facilmente transformada em expressões regulares, que são usadas no mecanismo de detecção deste projeto, o que será explicado com mais detalhes na próxima seção.

3.4 Detector de Intrusão

O módulo repensável pelo mecanismo de detecção de intrusão é implementado usando quatro classes: *Matcher*, *HeaderMatcher*, *PayloadMatcher* e *NonPlayloadMatcher*. A classe principal é o *Matcher* que chamará as outras classes para realizar a detecção. Como ilustrado na Figura 3.7, o processo inicia pelo *Matcher* que começa a detecção pelo cabeçalho. Caso todas as assinaturas do cabeçalho combine com o pacote, o processo continua com o *Non-Payload Matcher* que procura assinaturas nos dados do pacote que não fazem parte do *payload* (ex: Cabeçalho IP e TCP). Caso as assinaturas casem com os dados do pacote a detecção continua para o ultimo item que é o *PayloadMatcher*. Ele irá procurar no payload do pacote por assinaturas e em caso de casamento positivo é o registrado o resultado de acordo com as configurações da regra. Este procedimento é repetido para cada regra carregada no *bolt* e cada pacote será processado por todas as regras.

Após todo o procedimento, em caso de sucesso, o resultado do processamento é um objeto do tipo *Match* (Figura 3.8). Ele será registrado usando o módulo de persistência.

3.4.1 HeaderMatcher

A primeira etapa do motor de detecção do Storm IDS acontece no *HeaderMacher* que irá casar as informações do cabeçalho da regra com o pacote. A primeira verificação é a do protocolo onde no pacote e na regra devem ser iguais. Depois são verificados os endereços IP e portas de destino e origem. Como as regras possuem mais de um endereço, eles são armazenados na estrutura da regra na forma de *hashmaps* para agilizar a busca dos endereços do pacote a ser analisado. Caso o campo *direction* seja para ambas as

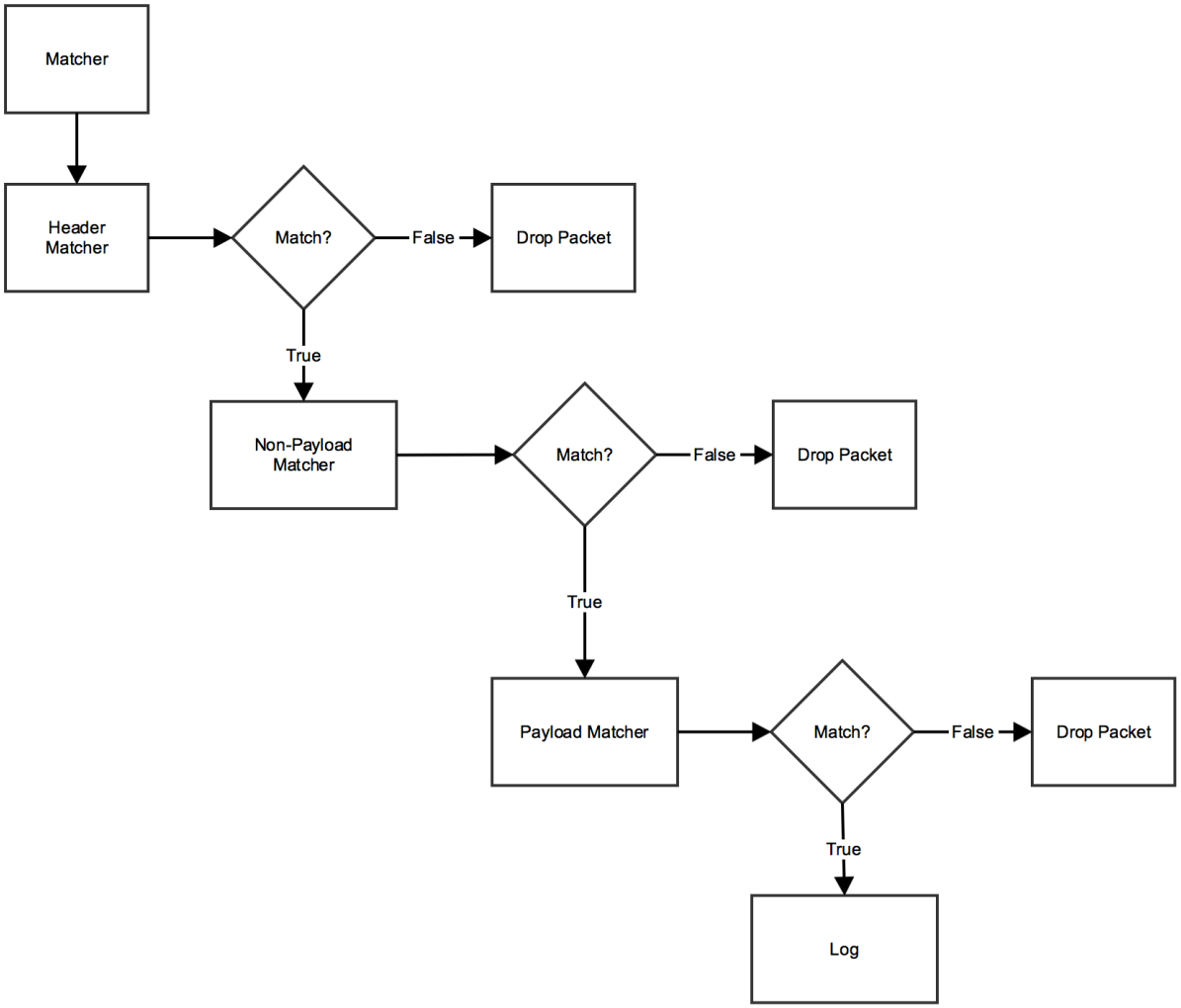

Figura 3.7: Fluxo do mecanismo de detecção de intrusão do Storm IDS.

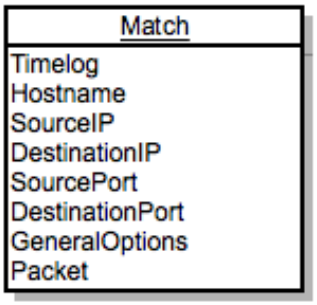

Figura 3.8: Diagrama de classes da estrutura de dados que representa o resultado da detecção que será salvo no banco.

direções, os endereços de destino e origem são trocados para se fazer a verificação na

direção contrária.

3.4.2 NonPayloadMatcher

Nesta etapa são verificados os campos do pacote que não fazem parte do payload. Como estes campos são muito específicos e possuem operações mais complexas do que simples comparação (maior, maior ou igual, menor, menor ou igual, negação, entre, etc) cada opção apresentada nas Tabela A.3 possui um método especifico para seu tratamento.

3.4.3 PayloadMatcher

Esta é a ultima etapa da detecção. No *PayloadMatcher* são procuradas assinaturas no campo de dados do pacote. Como esse campo pode ser muito grande, são utilizadas expressões regulares, previamente montadas no parser das regras do Snort, para se procurar padrões no *payload* do pacote.

O primeiro passo é decodificar os dados armazenados em base64 do pacote para uma string com codificação de caracteres ISO 8859-1. A grande vantagem do ISO 8859-1 é sua capacidade de mapear todos os bytes de 0x00 a 0xFF como um carácter e isso permite a utilização de *regex* em vetores de bytes. Depois é chamado o método *match* da instância da classe *Pattern* gerado na criação de acordo com os dados das opções de *payload* da regra.

3.4.4 Detector de intrusão do Storm IDS e do Snort

A lógica por trás do detector de intrusão do Storm IDS e do Snort são muito parecidas. Ambos usam uma cadeia para percorrer a estrutura de uma regra, começando do cabeçalho e partindo para as opções, parando a execução quando alguma opção não é combinada com o pacote [38]. Este processo evita com que processamentos mais demorados, como o casamento de padrão na carga do pacote, sejam feitos desnecessariamente. A grande diferença entre os dois sistemas detectores de intrusão é que o Storm IDS é um sistema distribuído. O Snort utilizada de paralelismo para agilizar a detecção de intrusão, entretanto ele está limitado ao poder de processamento de uma máquina apenas, enquanto o Storm IDS é capaz de executar em várias máquinas.

3.5 Persistência dos alertas

Ao iniciar, o *bolt LogMatches* é criada uma conexão com o banco de dados Cassandra e salva a sessão no atributo *session*. Após a detecção, a lista de alertas gerados é enviada

para o *LogMatches*. Esta classe possui um método chamado *saveMatches* que é capaz de salvar de maneira assíncrona todos os alertas recebidos, permitindo que o cassandra realize o processo de inserção no banco de dados seja mais eficiente possível [42] e antes do fim da execução é verificado se os dados foram salvos. Portanto, com a sessão que foi criada durante inicialização do *bolt* uma *query* para cada alerta é executada e os dados são salvos de maneira rápida.

3.6 Visualizador

Este é o módulo do Storm IDS responsável por mostrar ao usuários os alertas gerados pelo detector de intrusão. Foi criando um *web service* usando o framework Spring [41]. O visualizador ler os dados salvos no banco de dados e retorna ao usuário, ordenando-os pelo mais recente. Ele também permite a busca por alertas específicos. A Figura 3.9 mostra a tela do visualizador.

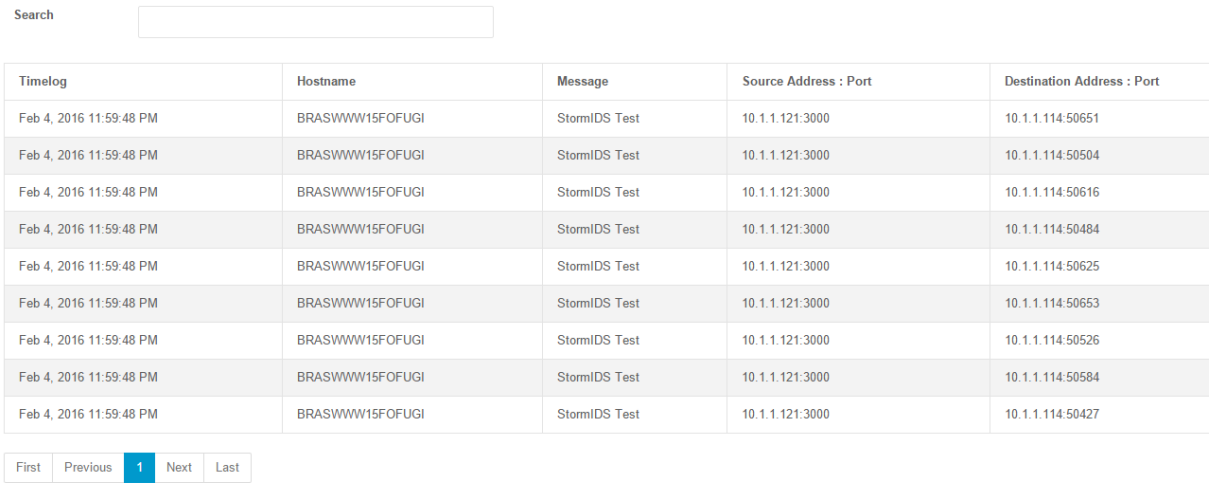

Figura 3.9: Visualizador de alertas.

3.7 Considerações Finais

Neste capítulos vimos os principais módulos do Storm IDS e como ele interagem. Também foi explicada a arquitetura por trás desse projeto e como os módulos se encaixam nos dois software que compõem o sistema, o Resource Monitor e o Storm Engine. No próximo capítulo serão mostrados os testes realizado neste projeto e analisados seus resultados.

Capítulo 4

Testes, Resultados e Análise

O objetivo principal dos testes demostrados a seguir é verificar o funcionamento do Storm IDS e validar se ele é capaz de detectar intrusões. Sendo assim, serão realizados testes unitários para cada módulo (Captura e Preprocessamento de Pacotes, Parser das Regras do Snort, Detector de Intrusão e Persistência) e teste de integração gerando pacotes que simulam ataques e verificando os resultados.

4.1 Testes Unitários

Antes de testar o sistema como um todo, foram realizado testes em cada módulo da aplicação para garantir seu funcionamento.

4.1.1 Módulo de Captura e Preprocessamento de Pacotes

Este é o módulo responsável por capturar os pacotes que serão enviados para a fila de mensagens. Para testar esse módulo foi construída uma aplicação web que consiste de um método *GET*, responsável por retornar uma mensagem com o texto "test"acompannado do horário da chamada. Foram utilizados um computador e um *smartphone* para realizar o teste, sendo que o computador foi o servidor e o *smartphone* o cliente. No servidor foi iniciado o *Resource Monitor* para capturar os pacotes.

Para facilitar a leitura dos testes, foi aplicado um filtro Berkeley Packet Filter (BPF) com as primitivas “(src 10.0.1.2 && dst 10.0.1.5) || (src 10.0.1.5 && dst 10.0.1.2)” que diz ao pcap4j para ler apenas os pacotes que pertencem a comunicação entre as aplicações que participaram do teste.

Na Figura 4.1 podemos ver a requisição “GET /test “que foi capturada utilizando o *Resource Monitor*. Na Figura 4.2, é possível verificar a resposta com a mensagem “Test 2015/12/06 14:40:01” no final do pacote.

```
IP origem = 10.0.1.2
IP destino = 10.0.1.5
¸èV=    84£  t§{□E□-  ï@□@  U
□
□ Í1  p ÄÜ  M6*   ?ñ□□
    X ÚÍØGET /test HTTP/1.1
Host: 10.0.1.5:8080
Cache-Control: max-age=0
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13B143 Safari/601.1
Accept-Language: en-us
Accept-Encoding: gzip, deflate
DNT: 1
```

Figura 4.1: Requisição da mensagem de teste ao servidor.

```
IP origem = 10.0.1.5
IP destino = 10.0.1.2
4£  t§{¸èV= 8□E□□Lß @□@ E
□
□   Í1  M6¸p ÆU     2â□□
 ÚÍÞ    XTest 2015/12/06 14:40:01
```

Figura 4.2: Resposta do servidor.

Observa-se nas imagens que o resultado demonstra a correta implementação da funcionalidade.

4.1.2 Módulo de Parse das Regras do Snort

Este é o módulo responsável por ler as regras do Snort e processá-las para serem usadas posteriormente na detecção de intrusão. Para testá-lo foi utilizada a ferramenta JUnit [5], responsável por auxiliar a geração de testes unitários nas classes envolvidas no processamento das assinaturas. Sendo assim, foram testadas as classes *Header*, *GeneralOptions*, *PayloadOptions* e *NonPayloadOptions*. A Figura 4.3 ilustra os resultados dos testes no JUnit.

Header Para testar a classe *Header*, foi realizado um teste simples. Utilizou-se um cabeçalho com o texto "alert tcp $HOME_NET any -> 192.168.1.0/24 111"e chamou-se o método *parse()* que é capaz de ler o cabeçalho e salvar os dados nos atributos da classe. Em seguida, verificou-se que os dados coletados estavam de acordo com a regra, demonstrando que a funcionalidade foi implementada corretamente.

GeneralOptions Para testar a classe *GeneralOptions*, usou-se a regra responsável por descobrir tentativas de ataques LFI [22], possuindo as opções gerais *metada*, *classtype*, *sid* e *rev*:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC
/etc/passwd"; flow:to_server,established; content:"/etc/passwd"; nocase;
```

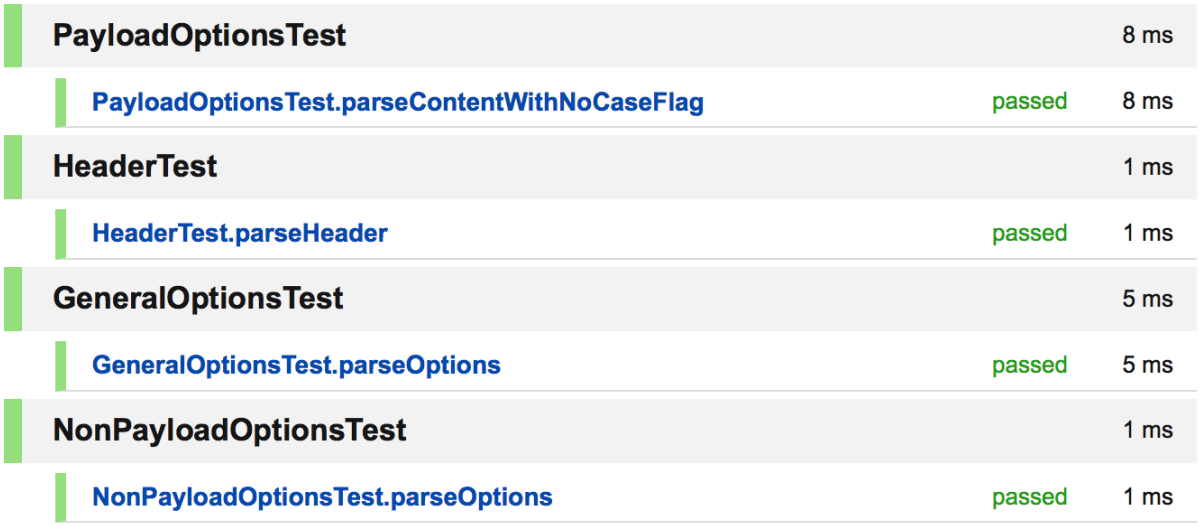

Figura 4.3: Resultado dos testes do módulo de parse das regras do snort no JUnit.

`http_uri; metadata:service http; classtype:attempted-recon; sid:1122; rev:8;)`

Então, chamou-se o método *parse()* que é capaz de ler a regra de forma completa e armazenar as opções nos atributos da classe. Em seguida, verificou-se que todos os campos referentes as *General Options* foram lidos corretamente. O resultado demonstra a correta implementação da funcionalidade.

PayloadOptions Para testar a classe *PayloadOptions*, usou-se a mesma regra acima. Usando-se o método *parse()*, a regra foi processada e foi gerada a expressão regular

`(?i)(.|\n|\r\n){0,}/etc/passwd`

que é capaz de identificar o padrão "/etc/passwd" independente do texto estar em caixa alta ou baixa. Este resultado demonstra a correta implementação da funcionalidade.

NonPayloadOptions Para testar a classe *PayloadOptions*, usou-se a mesma regra anterior adicionando os campos "(fragbits:M; fragoffset:0; ttl:>=5;tos:4; id:31337; ipopts:lsrr; fragbits:MD+; flags:SF,CE;)"". Posteriormente, usou-se o método *parse()* e verificou-se que os dados foram processados corretamente, demonstrando a correta implementação da funcionalidade.

4.1.3 Módulo de Detecção de Intrusão

Este é módulo que implementa a detecção de intrusão. Para testá-lo foi usada a mesma estratégia do módulo anterior. Foram criados testes para as classes *HeaderMatcher*, *PayloadMatcher*, *NonPayloadMatcher*. Todos os teste descritos à seguir foram realizados com sucesso. A Figura 4.4 ilustra os resultados dos testes no JUnit.

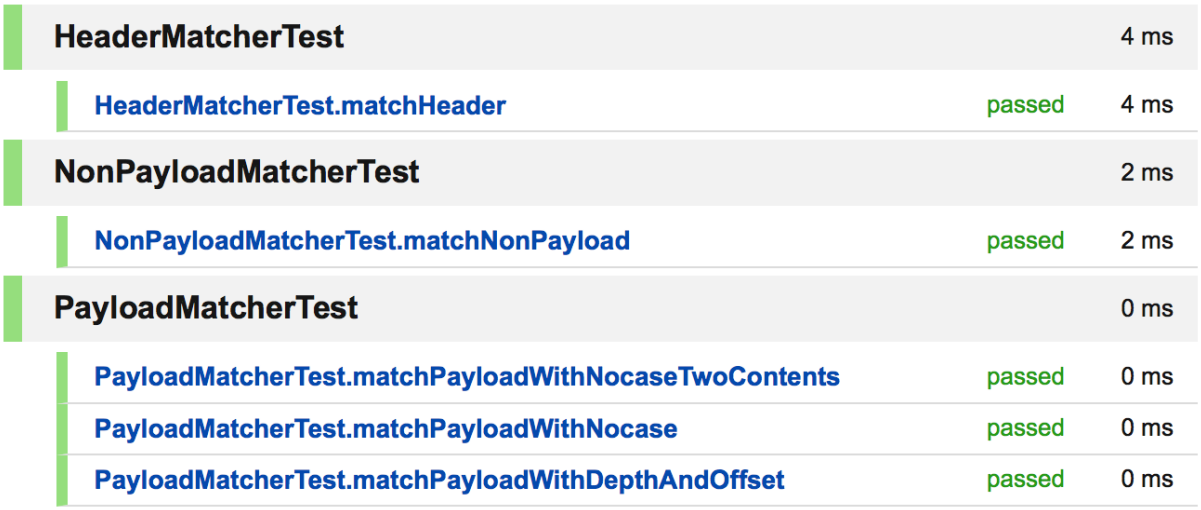

Figura 4.4: Resultado dos testes do módulo de detecção de intrusão no JUnit.

HeaderMatcher Para testar a classe *HeaderMatcher*, usou-se um pacote com protocolo TCP, endereço da fonte 10.0.1.5:433 e endereço de destino 192.168.1.5:111. Estes dados foram utilizados como entrada do método *match()* da classe junto com o cabeçalho de regra do Snort "alert tcp $HOME_NET any -> 192.168.1.5 111" onde em $HOME_NET temos o endereço IP 10.0.1.5 e constatou-se que o padrão foi encontrado demonstrando que a funcionalidade foi implementada corretamente.

PayloadMatcher Para testar a classe *PayloadMatcher*, realizou-se três testes que criam um objeto do tipo *PayloadOptions* com as especificações do conteúdo que será procurado em uma sequência de caracteres específicas.

O primeiro teste procurou-se o padrão "/../"no texto "kdddddnfcsdcdcdscsdcdABCcxzczxczxczcxczx /../". Foi configurado uma profundidade de quarenta caracteres e um ponto de início após cinco caracteres que gera a expressão regular:

```
^(.|\n|\r\n){0,}.{5,40} /../
```

No segundo, testou-se a funcionalidade *nocase* onde não há diferenciação de letras minúsculas e maiúsculas. Em seguida, foi configurado o padrão "def "para ser

procurado no texto “kdddddnfcsdcdcdscsdcdABCcxzczxczxczcxczxDEF ”com profundidade de quarenta caracteres e ponto de in[icio após cinco caracteres, gerando a expressão regular:

```
(?i)^(.|\n|\r\n){5,40}def
```

No terceiro teste é verificada a combinação de dois conteúdos que serão casados com os dados. Para isso, configurou-se dois *contents* no objeto da classe *PayloadOptions*. O primeiro conteúdo é igual ao do segundo teste e o segundo irá procurar o padrão “ABC”com uma distância de pelo menos um carácter após o primeiro *content*. A combinação gera a expressão regular

```
(?i)^(.|\n|\r\n){5,40}def(?i)(.|\n|\r\n){1,}ABC
```

a qual foi aplicada ao texto “kdddddnfcsdcdcdscsdcdABCcxzczxczxczcxczxDEF ABC”. É importante notar que o a sequência “ABC” aparece duas vezes no texto, porém ela só pode ser ativada se estiver após a sequência “DEF”com uma distância de pelo menos um carácter.

Em todos os testes o padrão foi encontrado, logo o resultado demonstra que a funcionalidade foi implementada corretamente.

NonPayloadMatcher Para testar a classe NonPayloadMatcher, usou-se um pacote com o protocolo TCP com todos os campos preenchidos de acordo com as seguintes opções de *non payload* do Snort: “(fragbits:M; fragoffset:0; ttl:>=5;tos:4; id:31337; ipopts:lsrr; fragbits:MD+; flags:SF,CE;)”. Estes dados foram utilizados como entrada do método match() e ao rodar o teste, verificou-se que o padrão foi encontrado o que demonstra que a funcionalidade foi implementada corretamente.

4.1.4 Módulo de Persistência

O último módulo testado foi o componente responsável por registrar as informações obtidas ao detectar uma intrusão. Neste módulo os alertas gerados pelo bolt *NetworkDataBolt* são salvos em uma instância do banco de dados Cassandra. Criou-se um teste unitário onde é criado uma lista contendo um objeto do tipo *Match* e enviado ao método *execute()* do bolt *LogMatchesBolt*. Logo após a execução deste procedimento, criou-se uma *query* selecionando o alerta que foi salvo e verificou-se que todos os dados estão de acordo com os previamente configurados. Este resultado demonstra a correta implementação da funcionalidade.

4.2 Testes de Integração

Nesta seção serão detalhados os testes de integração utilizados para demonstrar o funcionamento do Storm IDS.

4.2.1 Caso de Uso

Foi criado um ambiente de teste composto de dez computadores, todos possuindo um Intel Core i7 de quarta geração com 3.6GHz como processador, memória RAM de 16GB 1600MHz DDR3 e SSD de 256GB. O primeiro membro desta configuração é o "Atacante ", que foi responsável por gerar tráfego "malicioso" para as "vítimas". Atuando como *Vítimas* temos dois computadores monitorados pelo Resource Monitor que enviaram os seus pacotes para o Kafka. O quarto componente do ambiente foi designado para executar o Kafka, atuando como fila de mensagem. Os cinco próximos itens formam nosso *cluster Storm*, que foi responsável por executar o Storm Engine. O último membro é o nó responsável por exdcutar uma instância do Cassandra, sendo esse o banco de dados utilizado. Esta configuração pode ser visualizada na Figura 4.5

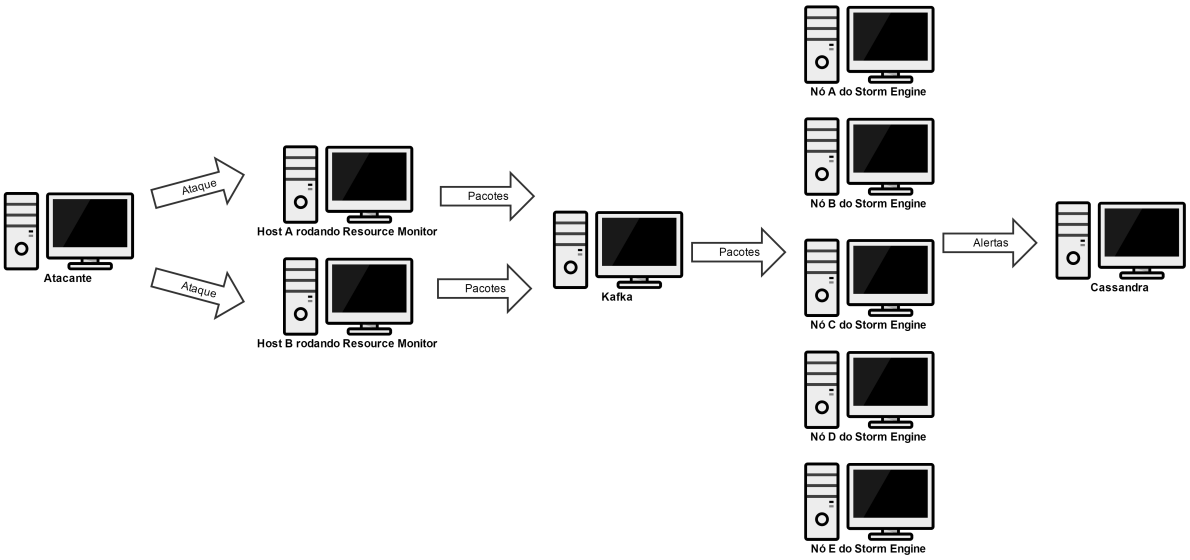

Figura 4.5: Ambiente de testes.

Para cada teste realizado esta configuração foi modificada possibilitando identificar o desempenho do Storm IDS em diferentes cenários.

4.2.2 Cenários

Foram implementadas dois cenários de testes.

Testes de Processamento Utilizado para medir a latência do Storm IDS gerar alertas passando por todos os estágios. Portanto, foi medido o tempo que demora para um ataque realizado ser alertado e o registro ser salvo no banco de dados.

Foram criadas duas pequenas aplicações que geram tráfego entre si, onde o servidor envia uma mensagem TCP, contendo o texto "StormIDS Test" mais bytes aleatórios, e o cliente responde dizendo que recebeu a mensagem. O servidor pode ser configurado para gerar uma quantidade X de pacotes com a mensagem e aceita conexões de diferentes clientes. Já o cliente pode ser configurado para executar em N *threads*. Desta forma foi possível configurar diferentes taxas de pacotes por segundo gerados para o Storm Engine.

Ao iniciar os clientes, foram gerados pacotes durante um período T de tempo, e ao final do envio, foi obtido a taxa de pacotes por segundo gerada.

Na parte do Detector de intrusão, foi criada a regra:

```
alert tcp $HOME_NET any -> any any (msg:"StormIDS Test";  flow:
established, to_server; content: "StormIDS Test";)
```

Esta assinatura alerta todos os pacotes que contém o texto "StormIDS Test"vindo do servidor de teste. Então foram gerados X pacotes "maliciosos" e medido quanto tempo após todos os pacotes serem enviados o StormIDS demorou para alertá-los.

Foram realizados testes com o Storm Engine possuindo um e dois nós. Para cada cenário foram executados dois clientes gerando tráfegos de 1Mbps, 3.5Mbps, 7Mbps, 14Mbps, 28Mbps, 56Mbps, 102Mbps e 204 Mbps.

Além da assinatura de teste, os testes foram realizados com todas as regras ativas disponibilizadas pelo Snort na versão 2.9.7.6, logo cada pacote foi processado por pouco mais de 7000. Posteriormente este número foi reduzido para 1000 para identificar o impacto do número de regras na latência do StormIDS.

Foram capturados apenas os pacotes vindo do servidor de teste para facilitar a leitura dos dados. Para cada pacote "infectado" foram gerados em média mais 2 pacotes, referentes a confirmações de envios realizadas pelo protocolo TCP, e estes pacotes também foram processados pelo Storm IDS.

Testes de Qualidade Foram realizado quatro ataques conhecidos e verificado se o Storm IDS é capaz de detectá-los. Também foi verificado se o Storm IDS gera falsos positivos e foi medida a relação entre falsos positivos e ataques reais.

Estes testes são baseado em [1] que realizou os mesmos testes nos IDSs Snort e Suricata.

Os ataques testados foram:

Local File Inclusion (FLI): É um ataque comum em *scripts* PHP. Este consiste na inclusão de classes ou funções para uso futuro podendo ser implementadas usando os métodos PHP include(), require(), include_once(), require_once(), etc [22]. O processamento destes arquivos incluídos podem gerar alguns problemas. Alguns *websites* proveem navegação pela inclusão de arquivos de acordo com o parâmetro da URL. Por exemplo, *http://example.com/index.php?site=home.hmtl*. Existe a possibilidade que esta variável do site seja o nome do arquivo que o atacante quer incluir, então ele pode manipular o valor da variável para explorar as funcionalidades do site ou extrair dados que ele não deveria ter acesso [22].

Para testar essa vulnerabilizada foi executado uma chamada http que tenta realizar o ataque descrito acima e verificado se o alerta esperado foi gerado.

Full SYN scan: Foi utilizado o software Nmap [27] para realizar uma varredura completa na rede monitorada. Este programa é capaz de descobrir aplicações disponíveis em uma rede.

Para realizar este teste foi feita uma varredura no computador monitorado.

SQL Injection: Estes são ataques do tipo de injeção de código, no qual dados providos pelo usuário são incluídos em uma *querry* SQL de uma maneira que parte da entrada do usuário é tratada como código SQL [18]. Esta vulnerabilidade pode ser explorada permitindo o atacante enviar comandos SQL direto para o banco de dados.

Neste teste foi realizada uma chamada HTTP que simula uma injeção de SQL.

Netcat reverse shell: Netcat é uma que cria conexões TCP ou UDP bilaterais possuindo muitas opções de *debugging* e exploração [20]. Aproveitando-se das funcionalidads do Netcat, foi criada a técnica de *reverse shell* onde a vítima conecta com o atacante e entrega o *command shell* [31]. Entretanto, realizar o comando na vítima não é tão indicado e pode ser necessário usar de algumas técnicas, como fazer um *upload* de um arquivo PHP com o procedimento que realiza o *reverse shell* e acessá-lo através de um *browser* [31].

4.2.3 Resultados

Testes de Processamento Foram realizados cinco testes principais neste cenário. Cada teste foi realizado com um número de nós diferentes no *cluster*, sendo um no primeiro teste, dois no segundo e assim por diante, finalizando com cinco nós no ultimo

teste. Para cada nó foram instanciados um *KafkaSpout*, um *MultiplexerBolt*, um *PrettyPrinterBolt*, seis *NetworkDataBolts* e um *LogMatchesBolts* e cada pacote foi processado por pouco mais de 7000 regras.

Nos primeiros testes realizados foi medida a quantidade de tempo em segundos que os pacotes enviados demoram para serem alertados e se a quantidade de alertas correspondem com o esperado utilizando apenas um e dois nós no cluster Storm. A primeira medida pode ser visualizada na Figura 4.6, onde a parti de 7.2Mbps a aplicação demorou 20 segundos para gerar os alertas após o fim da transmissão dos pacotes. Já quando o número de pacotes aumentou para 21.6Mbps, o Storm IDS teve uma latência de 255 segundos para gerar os alertas. Estes resultados também podem ser visualizados na Tabela B.1. A segunda medida, ilustrada na Figura 4.7, mostra que o programa começou a perder alertas a parti de 7.2Mbps, porém esse número foi considerado significativo quando eram esperados 384000 alertas pois só foram alertados 65% dos pacotes com a assinatura de teste.

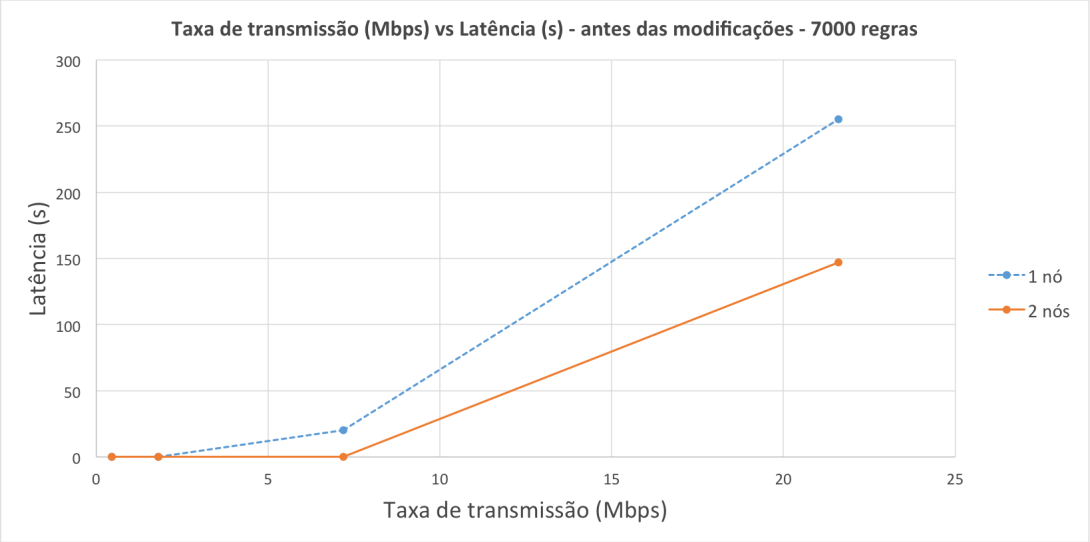

Figura 4.6: Tráfego gerado por latência do alerta antes das modificações no Storm IDS e utilizando 7000 regras.

Utilizando dois nós no cluster Storm, houve-se uma melhora significativa dos resultados. Na primeira medida, podemos ver na Figura 4.6 que o atraso foi evidente a partir dos 21,6Mbps por segundo, onde foi notada uma latência de 145 segundos, 110 segundos menor do que no primeiro cenário. Esta redução acontece porque o processamento foi distribuído em mais *threads*. Já a Figura 4.8 mostra que a quantidade de alertas despercebidos também foi significativamente menor, sendo alertados 99.99997% dos casos, tendo em vista que a lista de pacotes processada por cada *thread* era menor, reduzindo a chance de exceções por falta de memória.

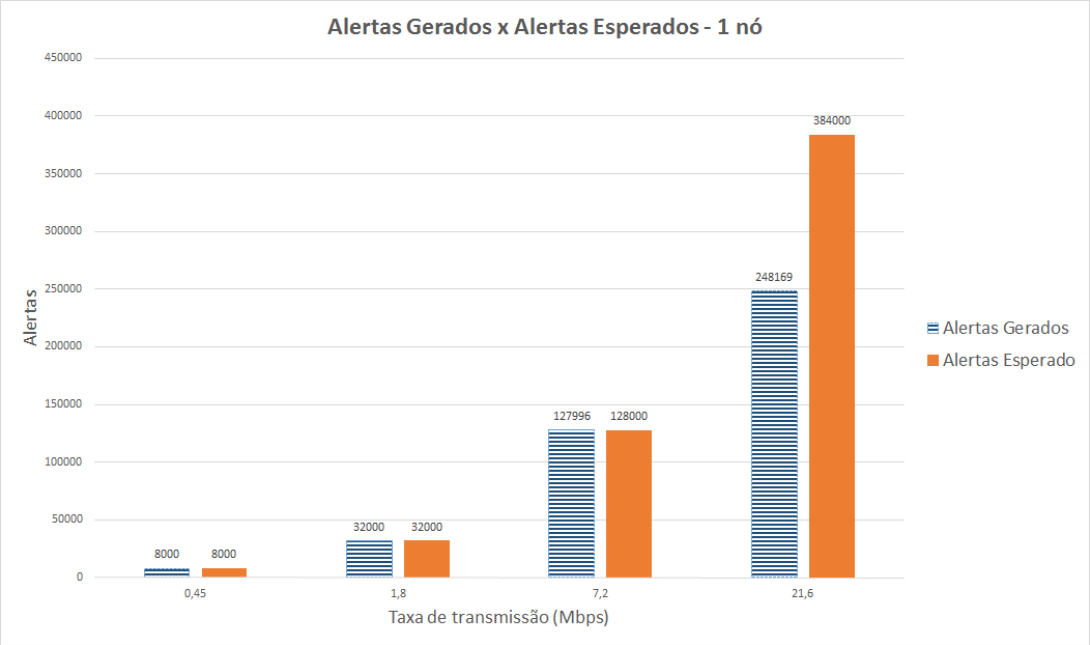

Figura 4.7: Relação entre alertas gerados e alertas esperados com um nó no Storm Engine.

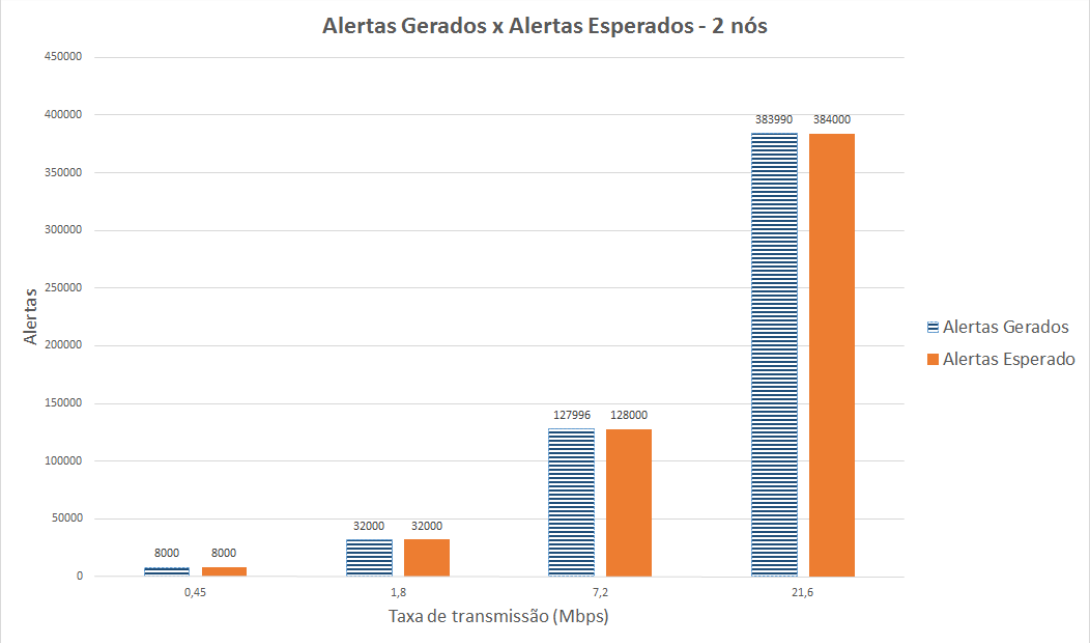

Figura 4.8: Relação entre alertas gerados e alertas esperados com dois nós no Storm Engine.

Após estes testes, foram identificados, pontos que poderiam melhorar a performance do software e ajustes foram realizados. Na primeira implementação do Storm IDS, todas as mensagens eram trocadas no formato JSON e isso mostrou-se um gargalo

para a aplicação. Sendo assim, removeu-se todas as conversões de objetos para JSON, realizando a troca de mensagens com sequências de bytes. Os mesmos testes foram realizados após as mudanças e foram gerados os seguintes resultados:

Em todos os testes tivemos melhoras nos resultados. O Storm IDS conseguiu suportar uma quantidade consideravelmente maior de dados, porém, a biblioteca jpcap não conseguiu capturar os pacotes ao tentar-se gerar um tráfego acima de 204Mbps, induzindo o Resource Monitor a falha. No primeiro e segundo teste a detecção de intrusão não mostrou nenhum atraso até 14Mbps. Entretanto, com uma vazão de 28Mbps os primeiros sinais de latência foram apresentados. Com o aumento da taxa de bits por segundo, o atraso aumentou e, como esperado, no primeiro teste o atraso foi maior do que os demais. Entretanto, com o aumento gradual da quantidade de unidades de processamento no *cluster*, o Storm IDS conseguiu melhorar seu desempenho de maneira significativa tendo no último teste uma latência de 103 segundos. A Figura 4.9 ilustra estes resultados. Estes resultados também podem ser visualizados na Tabela B.2.

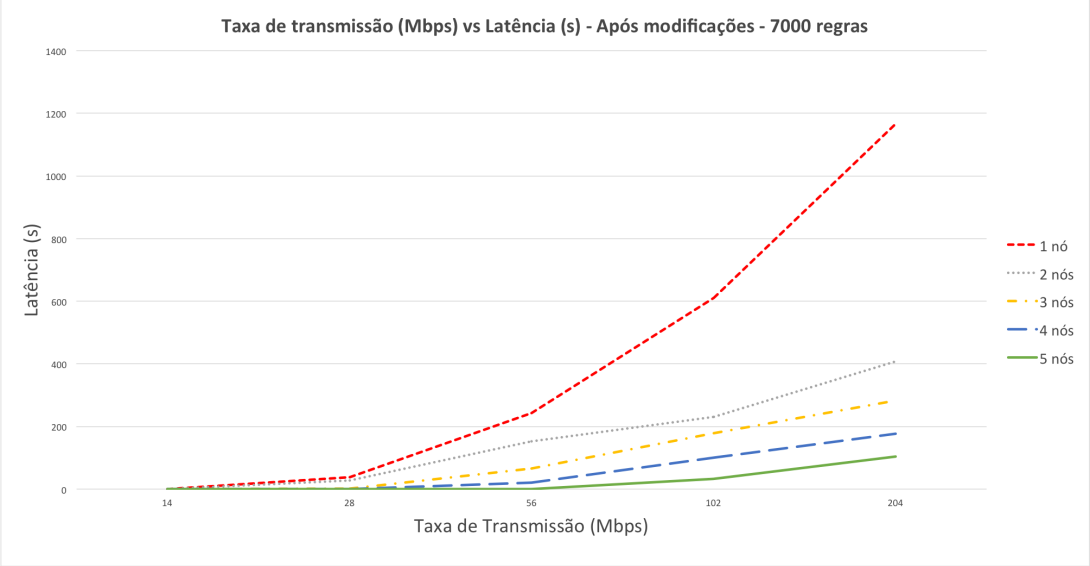

Figura 4.9: Tráfego gerado por latência do alerta após a modificações no Storm IDS e utilizando 7000 regras.

Considerando a latência observada no teste anterior, foi realizado mais um teste, com apenas dois nó no cluster, com o número total de regras reduzido para 1000. Como pode-se observar na Tabela B.3 a latência da detecção foi insignificante, sendo de 1 segundo quando a vazão era 102Mbps e 204Mpbs.

Testes de Qualidade Foram realizados quatro testes que apresentaram os seguintes resultados:

Local File Inclusion (FLI) Para simular este ataque foi realizada a seguinte requisição na HTTP [1]:

```
echo "GET /index.php?page=../../../etc/passwd HTTP/1. 1\r\nHost:
127.0.0.1\r\nUser-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1
; en-US; rv:1.7.5) Gecko/20041202 Firefox/1.0\r\n\r\n" | nc
10.0.1.5 80
```

O alerta esperado foi gerado, como pode ser visto na Figura 4.10. Porém outros 276 alertas foram gerados.

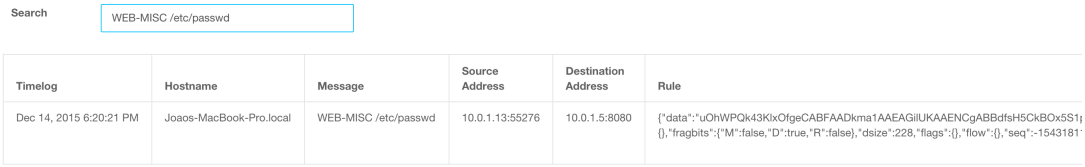

Figura 4.10: Captura do ataque FLI.

Full SYN scan: Para simular este ataque foi realizada a seguinte comando do nmap [1]:

```
sudo nmap -sS -p- 10.0.1.5
```

Contudo, foram gerados mais de 70000 alertas durante o escaneamento das portas, porém nenhum referente ao ataque.

SQL Injection (UNION SELECT): Para simular este ataque foi realizada a seguinte requisição na HTTP [1]:

```
echo "GET /form.php?q=1+UNION+SELECT+VERSION%28%29 HTTP/1.1\r\nHost:
127.0.0.1\r\n\r\n" | nc 10.0.1.5 80
```

Foram gerados 232 alertas, porém nenhum referente ao ataque.

Netcat reverse shell: Para simular este ataque foi acionado o seguinte comando [1]:

```
echo "/bin/sh" | nc 10.0.1.5 22
```

O alerta esperado foi gerado, como pode ser visto na Figura 4.11. Porém outros 146 alertas foram gerados.

4.3 Análise

Como esperado o StormIDS foi capaz de escalar ao aumentar o número de nós do Storm Engine. Porém, ele demostrou latência para gerar alertas ao final da transmissão de

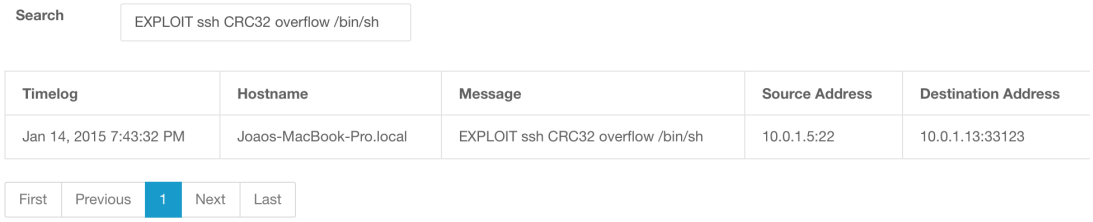

Figura 4.11: Captura do ataque *Netcat reverse shell*.

uma grande quantidade de pacotes e algumas limitações com o aumento do número de pacotes enviados por segundo. Como pode ser notado na Figura 4.12, o maior gargalo da topologia é o *NetworkMonitorBolt* que deve processar mais de 7000 regras por pacote. A cor vermelha indica que a latência estar muito alta e que os demais *bolts* estão com bem menos processamento. Outro gargalo é codificação e decodificação de e para base64 da carga dos pacotes. Este processo pode levar a estouros de memória e lançar exceções no StormIDS, o fazendo perder pacotes. Este foi o processo que levou o primeiro cenário de testes de processamento a gerar apenas 65% dos alertas esperados.

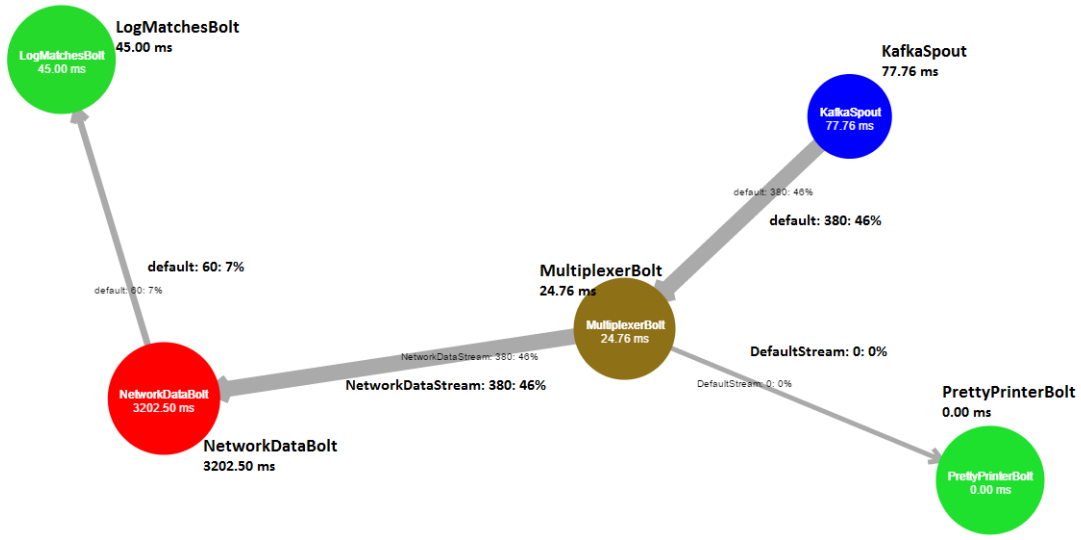

Figura 4.12: Pico de processamento dos pacotes durante os testes.

Tendo em vista este resultado, foram retirados todos os procedimentos desnecessários para transformação dos dados em JSON e a carga do pacote para base64. Isto otimizou significativamente a *performance* do Storm IDS, sendo capaz de processar até 200Mbps

de tráfego. Entretanto, a latência continuou consideravelmente alta para uma vazão volumosa.

Também deve ser notado que com o aumento do número de unidades de processamento o sistema conseguiu melhorar seu desempenho de forma significativa, demonstrando que aumentando o número de *bolts* é possível alcançar resultados aceitáveis.

Apesar das melhorias significativas, o *NetworkMonitorBolt* continuou sendo um gargalo, pois muitas regras são processadas para cada pacote. Em seguida, foi realizado mais um teste reduzindo o número de regras de 7000 para 1000. Como pode ser notado na Figura 4.13, o tempo de processamento no *NetworkMonitorBolt* foi muito menor e o Storm IDS foi capaz de rodar com uma latência aceitavel.

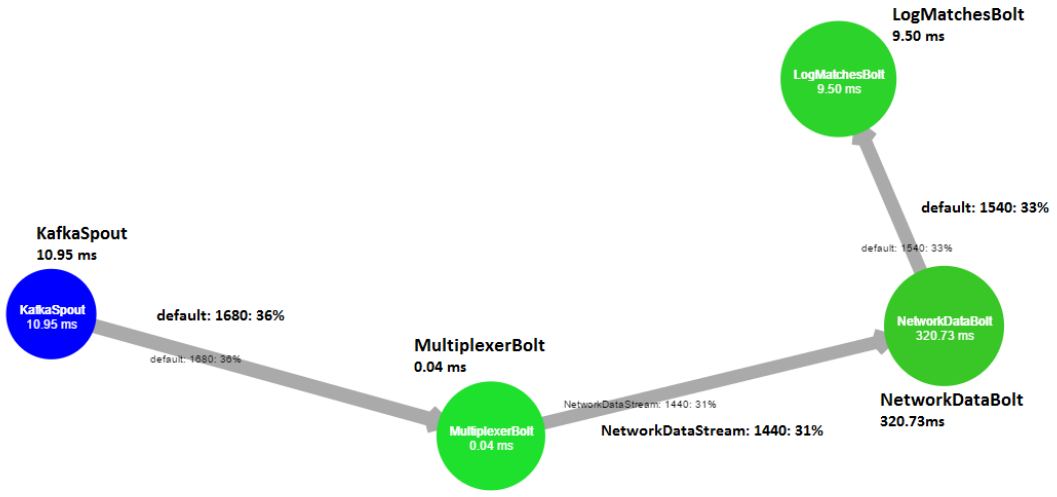

Figura 4.13: Pico de processamento dos pacotes durante os testes com menos regras.

Posteriormente foram realizados os testes de qualidade que demonstraram que o StormIDS é capaz de alertar ameaças reais, porém com um número alto de falsos positivos. Isto aconteceu porque a qualidade da detecção de intrusão está diretamente relacionada a qualidade das regras utilizadas. Antes da instalação de qualquer IDS, deve-se fazer um estudo rigoroso das assinaturas usadas para evitar que falsas ameaças sejam registradas.

Capítulo 5

Conclusão

O Storm IDS, um sistema detector de intrusão capaz de processar dados em tempo real de maneira distribuída foi projetado e implementado.

Este sistema é composto de dois componentes principais, o *Resource Monitor* e o *Storm Engine*, que combinados capturam pacotes de computadores monitorados, processam estes dados e geram alertas de atividades maliciosas. Para visualizar estes alertas, uma terceira aplicação foi desenvolvida, que consiste de um *web service* que ler as informações do banco de dados e retorna para uma página HTML.

Para desenvolver o Storm IDS foram projetados cinco módulos. O primeiro é o módulo de captura e preprocessamento de pacotes. Este é responsável por capturar os dados de rede do computador monitorado, processá-los e enviá-los para a fila de mensagens. O próximo componente é o *parser* das regras do Snort, que realiza a leitura das assinaturas disponibilizadas pela equipe de desenvolvimento do Snort e as processa para que sejam usadas na detecção de intrusão. O terceiro módulo do sistema é o detector de intrusão que é responsável pela leitura destas regras e as aplicam em todos os pacotes que são enviados ao *Storm Engine*. Após uma ameaça ser detectada, o módulo de persistência entra em ação e salva o alerta no banco de dados. Por último, há o módulo de visualização que realiza a leitura dos dados do banco de dados e mostra ao usuário.

No estado atual, o Storm IDS é capaz de detectar intrusões em ambientes distribuídos e escalar. Com o uso de tecnologias como o Storm, Kafka e Cassandra, o Storm IDS consegue aumentar sua capacidade de processamento apenas aumentando o número de nós rodando o sistema, pois o código é projetado para se adaptar a esta característica.

Entretanto, ainda existem algumas limitações quanto a sua capacidade de encontrar ameaças reais, pois esta eficiência está atrelada a qualidade das assinaturas que são usadas pelo sistema. Como a maioria das regras criadas para sistemas detectores de intrusões atualmente são desenvolvidas por pessoas, isso torna este processo lento e propenso a falhas.

Em conclusão, a arquitetura do Storm IDS pode ser fortemente vantajosa para ambientes em que deve ser garantida a segurança em diversos dispositivos diferentes, pois ela é capaz de detectar intrusões em ambientes distribuídos. Sua capacidade de distribuir o processamento aumenta muito sua escalabilidade e possibilita processamento de grande de quantidade de dados em tempo real, além de não ser um alvo para ataques DoS, como IDSs e *firewalls* tradicionais.

5.1 Trabalhos Futuros

Otimização da ferramenta Apesar da escalabilidade do Storm IDS, a otimização de seu algorítimo de detecção pode gerar muitos benefícios.

Como foi constatado nos testes, o maior impacto no desempenho do Storm IDS está no número de regras que são processadas em cada pacote. Para diminuir este número, é interessante agrupar regras por categorias e na captura destes pacotes identificar a categoria de cada pacote. Assim todos os pacotes somente serão processados pelas regras que são do mesmo protocolo. Por exemplo, se há regras específicas do HTTP apenas pacotes HTTP devem ser processados por estas regras.

Outra estratégia seria diminuir o número de regras processadas por *bolts*, combinando apenas 1000 regras por pacote em cada unidade de processamento. Para isto seria necessário distribuir o número total de regras para um número X de *bolts* onde cada um processaria uma faixa de assinaturas diferente. Também seria necessário criar um *bolt* específico que funcionaria como *switch* para distribuir os pacotes para as unidades de processamento corretas, assim cada pacote seria processado por X *bolts* diferentes. Esta abordagem pode diminuir a latência do Storm IDS, pois foi constatado que a quantidade de assinaturas tem um impacto muito grande no atraso ao gerar alertas, porém para suportar uma quantidade maior de regras seriam necessárias mais unidades de processamento, além do *bolt switch* poder se tornar um gargalo.

Extensão da ferramenta O Storm IDS é um NIDS que captura dados direito do *Host*. Com a extensibilidade do *Resource Monitor* e *Storm Engine*, pode-se desenvolver novos *plugins* para a criação de um HIDS atuando junto com o NIDS e melhorando a sua capacidade de detectar atividades maliciosas.

Suporte a regras de outros IDSs Para melhorar a eficiência da detecção de intrusões, seria interessante processar assinaturas de outros IDS, como o Suricata [13].

Criação automática de regras O processo de criação de regras atual é manual. O desenvolvimento de algoritmos de *machine learning* para criação de regras baseado em resultados de padrões conhecidos aumentaria consideravelmente a eficiência do IDS.

Referências

[1] Aldeid. Suricata-vs-snort. `https://www.aldeid.com/wiki/Suricata-vs-snort`. 41, 46

[2] A. S. Ashoor e S. Gore. Importance of intrusion detection system (ids). *International Journal of Scientific and Engineering Research*, 2(1):1–4, 2011. 4, 5

[3] R. G. Bace. *Intrusion Detection*. Macmillan Technical Publishing, 2000. x, 5

[4] A. Toshniwal et al. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. 13

[5] D. Saff et al. Junit. `http://junit.org`. 36

[6] E. Guillen et al. Inefficiency of ids static anomaly detectors in real-world networks. *Future Internet*, 7(2):94, 2015. 4

[7] J. Leibiusky et al. *Getting Started with Storm*. O'Reilly Media, Inc., 2012. 15

[8] M. M. Pillai et al. An approach to implement a network intrusion detection system using genetic algorithms. In *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '04, pages 221–221, Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists. 5, 6

[9] M. Zuckerberg et al. Facebook. `https://www.facebook.com`. 18

[10] T. Rabl et al. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, August 2012. 19

[11] W. Bulajoul et al. Network intrusion detection systems in high-speed traffic in computer networks. In *e-Business Engineering (ICEBE), 2013 IEEE 10th International Conference on*, pages 168–175, Sept 2013. 26

[12] Apache Software Foundation. Kafka. `http://kafka.apache.org/`. 17

[13] Open Information Security Foundation. Suricata. `http://suricata-ids.org`. 1, 50

[14] The Apache Software Foundation. Apache hbase. `http://hbase.apache.org/`. 19

[15] N. Garg. *Learning Apache Kafka, Second Edition*. Packt Publishing, 2nd edition, 2015. x, 17, 18

[16] P. T. Goetz e B. O'Neill. *Storm blueprints: Patterns for distributed real-time computation*. Packt Publishing Ltd, 2014. x, 14, 15, 16

[17] R. Graham. *FAQ: Network Intrusion Detection Systems*. 2000. 4

[18] William G.J. Halfond, Jeremy Viegas, e Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, Washington D.C., USA, March 2006. 42

[19] E. Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010. 18

[20] Hobbit. Netcat. `http://sectools.org/tool/netcat/`. 42

[21] A. Jain e A. Nalya. *Learning Storm*. Packt Publishing, 2014. x, 13, 14, 15, 16

[22] A. et al. Kapczyński. *Internet - Technical Developments and Applications 2*. Advances in Intelligent and Soft Computing. Springer Berlin Heidelberg, 2012. 36, 42

[23] H. Kozushko. Intrusion detection: Host-based and network-based intrusion detection systems, 2003. `http://www.buzzardsbranch.com/intrude/IntrusionDetectionPaper.pdf`. 4

[24] S. Kumar. *Classification And Detection Of Computer Intrusions*. Purdue University, 1995. 5, 6

[25] S Kumar e E H. Spafford. A Software Architecture to support Misuse Intrusion Detection. In *Proceedings of the 18th National Information Security Conference*, number 95 009, pages 194–204, 1995. 5

[26] A. Lakshman e P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. 18

[27] G. Lyon. Nmap. `https://nmap.org/`. 42

[28] N. Marz. Storm. `http://storm.apache.org`. vi, vii, 2, 13

[29] Microsoft. Microsoft iis. `https://www.iis.net`. 12

[30] R. Miller. Ballmer: Microsoft has 1 million servers. `http://www.datacenterknowledge.com/archives/2013/07/15/ballmer-microsoft-has-1-million-servers/`. 1

[31] S. Moon. Netcat tutorial – command examples on linux. `http://www.binarytides.com/netcat-tutorial-for-beginners/`. 42

[32] MySQL. Mysql. `http://www.mysql.com/`. 19

[33] N. Neeraj. *Mastering Apache Cassandra*. Packt Publishing, 2013. x, 19, 20

[34] J. P. Planquart. Application of neural networks to intrusion detection, 2001. 4

[35] The Netty project. Netty. http://netty.io/. 20

[36] R. U. Rehman. *Intrusion Detection Systems with Snort Advanced IDS Techniques Using Snort, Apache, MySQL, PHP, and ACID*. Prentice Hall, 2003. 6, 7, 8, 9, 10, 26

[37] M. Roesch. Snort. https://www.snort.org/. vi, vii, 1, 2, 6

[38] M. Roesch. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999. 8, 10, 33

[39] M. Roesch e The Snort Team. *Snort Users Manual*. Sourcefire, 2.9.7.3 edition, 2015. https://www.snort.org/. 11, 12, 28, 29

[40] S. Sanfilippo. Redis. http://redis.io/. 19

[41] Pivotal Software. Spring. https://spring.io/. 34

[42] R. Svihla. Cassandra: Batch loading without the batch keyword. https://medium.com/@foundev/cassandra-batch-loading-without-the-batch-keyword-40f00e35e23e#.anduivq4m. 34

[43] L. F. R. Taveira. *Monitoramento de Ambientes Computacionais Distribuídos em Tempo Real*. Universidade de Brasília, 2015. vi, vii, x, 2, 20, 21, 22, 24, 25, 26, 27

[44] Voldermort. Project voldermort. http://project-voldemort.com/. 19

[45] VoltDB. Voltdb. http://voltdb.com/. 19

[46] K. Yamada. Pcap4j, 2015. https://github.com/kaitoy/pcap4j. 28

[47] D. Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, West Lafayette, IN. 4

Apêndice A

Opções do Snort

Tabela A.1: General Options.

Palavra Chave	**Descrição**
msg	A palavra chave *msg* diz ao sistema de registro ou alerta qual a mensagem a ser mostrada.
reference	A palavra chave *reference* permite que seja incluída referencias a sistemas externo de identificação de ataques para que a regra tenha mais credibilidade.
gid	A palavra chave *gid* é usada para identificar qual parte do Snort gerou o evento quando uma regra especifica é acionada.
sid	A palavra chave *sid* é usada para identificar unicamente as regras do Snort
rev	A palavra chave *rev* é usada para identificar unicamente revisões de uma regra do Snort
classtype	A palavra chave *classtype* é usada para caracterizar a regra de acordo com o ataque que ela detecta.
priority	A palavra chave *priority* define o nível de severidade da regra.
metadata	A palavra chave *metadata* permite que o escritor de regras adicione informações extras sobre a regra.

Tabela A.2: Payload Options.

Palavra Chave	**Descrição**
content	A palavra chave *content* permite ao usuário configurar regras que podem procurar por um conteúdo especifico em no payload de um pacote.
rawbytes	A palavra chave *rawbytes* permite que as regras olhem os dados bruto do pacote, ignorando qualquer decodificação pode ter sido feita pelos preprocessadores.
depth	A palavra chave *depth* permite que o criador da regra defina a profundidade a qual deve ser feita a pesquisa pelo padrão especificado.
offset	A palavra chave *offset* permite que o criador da regra defina o ponto onde a pesquisa pelo padrão começará.
distance	A palavra chave *distance* permite que o criador da regra especificar o quão dentro do pacote deve ser ignorado antes de começar a procurar por um padrão especifico relativo ao fim do casamento de padrão anterior.
within	A palavra chave *within* certifica que no máximo N bytes estão entre os casamentos de padrão usando a palavra chave *content*.
content	A palavra chave *content* permite ao usuário configurar regras que podem procurar por um conteúdo especifico em no payload de um pacote.
rawbytes	A palavra chave *rawbytes* permite que as regras olhem os dados bruto do pacote, ignorando qualquer decodificação pode ter sido feita pelos preprocessadores.
depth	A palavra chave *depth* permite que o criador da regra defina a profundidade a qual deve ser feita a pesquisa pelo padrão especificado.
offset	A palavra chave *offset* permite que o criador da regra defina o ponto onde a pesquisa pelo padrão começará.
distance	A palavra chave *distance* permite que o criador da regra especificar o quão dentro do pacote deve ser ignorado antes de começar a procurar por um padrão especifico relativo ao fim do casamento de padrão anterior.
within	A palavra chave *within* certifica que no máximo N bytes estão entre os casamentos de padrão usando a palavra chave *content*.

Tabela A.3: Non-Payload Options.

Palavra Chave	**Descrição**
fragoffset	A palavra chave *fragoffset* permite a comparação do campo *fragment offset* do protocolo IP com um número decimal.
ttl	A palavra chave *ttl* é usada para checar o campo *time-to-live* do IP.
tos	A palavra chave *tos* é usada para checar o campo *TOS* do IP.
ipopts	A palavra chave *ipopts* é usada para checar se uma opção especifica está no campo *option* do IP.
fragbits	A palavra chave *fragbits* é usada para checar se as os bits *fragmentation* e *reserved* estão setados no cabeçalho IP
dsize	A palavra chave *dsize* é usada para checar o tamanho do payload do pacote.
seq	A palavra chave *seq* é usada para checar um número de sequencia especifico no protocolo TCP.
ack	A palavra chave *ack* é usada para checar por um número *acknowledge* especifico no protocolo TCP.
window	A palavra chave *window* é usada para checar um tamanho de janela especifico no protocolo TCP.
itype	A palavra chave *itype* é usada para checar um tipo especifico no protocolo ICMP.
icode	A palavra chave *icode* é usada para chegar um um código especifico no protocolo ICMP.
icmp id	A palavra chave *icmp id* é usada para checar um valor ID especifico no protocolo ICMP.
icmp seq	A palavra chave *icmp seq* é usada para checar um valor especifico de sequencia no protocolo ICMP.
ip proto	A palavra chave *ip proto* é usada para checar o valor no campo *protocol* no cabeçalho IP.
same ip	A palavra chave *same ip* é usada para checar se o ip da fonte é igual ao ip de destino.

Tabela A.4: Post-Detection Options.

Palavra Chave	**Descrição**
logto	A palavra chave *logto* diz ao Snort para logar todos os pacotes que ativar essa regra em um arquivo especial.
session	A palavra chave *session* é usada para extrair dados do usuário .
resp	A palavra chave *resp* é usada para fechar uma seção quando um alerta é acionado.
react	A palavra chave *react* permite ao usuário reagir ao trafego que combina com uma regra do Snort fechando a conexão ou enviando uma notificação .
tag	A palavra chave *tag* permite o usuário a logar mais informações do que apenas o pacote que acionou a regra.
activates	A palavra chave *activates* permite o usuário acionar outra regra quando um evento ocorre.

Apêndice B

Taxa de transmissão (Mbps) vs Latência (s)

Tabela B.1: Taxa de transmissão (Mbps) vs Latência (s) antes das modificações no Storm IDS e utilizando 7000 regras.

Taxa de transmissão (Mbps)	**1 nó**	**2 nós**
0.45	0	0
1.8	0	0
7.2	20	0
21.6	255	147

Tabela B.2: Taxa de transmissão (Mbps) vs Latência (s) após a modificações no Storm IDS e utilizando 7000 regras.

Taxa de transmissão (Mbps)	**1 nó**	**2 nós**	**3 nós**	**4 nós**	**5 nós**
1	0	0	0	0	0
3.5	0	0	0	0	0
7	0	0	0	0	0
14	0	0	0	0	0
28	38	28	1	0	0
56	243	152	65	21	0
102	610	231	179	101	33
204	1165	407	282	176	103

Tabela B.3: Taxa de transmissão (Mbps) vs Latência (s) com o número de regras reduzido para 1000.

Taxa de transmissão (Mbps)	**Latência (s)**
1	0
3.5	0
7	0
14	0
28	0
56	0
102	1
204	1